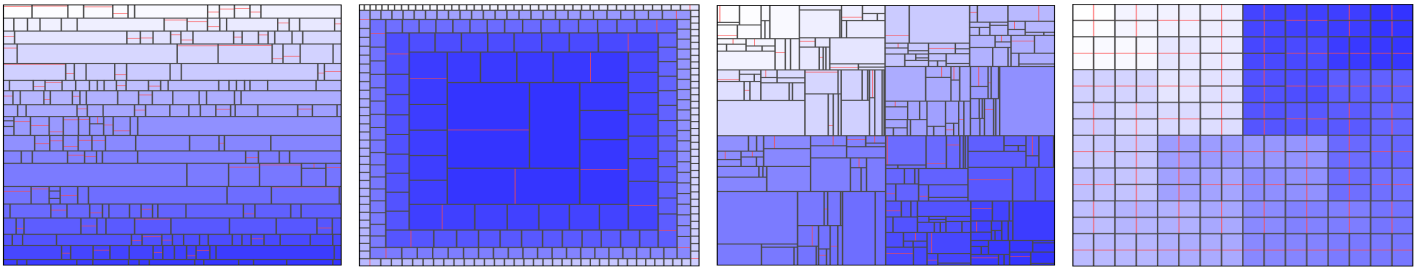
Capturing the Design Space of Sequential Space-Filling Layouts

Thomas Baudel and Bertjan Broeksema



(a)

Order:	data
Size:	data.Population
Chunking:	BestMinAspectRatio
Recurse:	$\#items > 2$
Phrase:	ZigZag

(b)

Order:	data.Population
Size:	data.Population
Chunking:	BestMinAspectRatio
Recurse:	never
Phrase:	Spiral

(c)

Order:	data
Size:	data.Population
Chunking:	PivotBySize
Recurse:	$\#items > 2$
Phrase:	WorstDiscontinuous

(d)

Order:	data
Size:	constant
Chunking:	HalfItemCount
Recurse:	$\#items \ \% \ 4 == 0$
Phrase:	Hilbert

Fig. 1: various space-filling layout examples: (a) strip (with recursion), (b) spiral, (c) pivot by middle. (d) hilbert phrasing, Chunks are separated by black lines, individual items by red lines.

Abstract—We characterize the design space of the algorithms that sequentially tile a rectangular area with smaller, fixed-surface, rectangles. This space consist of five independent dimensions: Order, Size, Score, Recurse and Phrase. Each of these dimensions describe a particular aspect of such layout tasks. This class of layouts is interesting, because, beyond encompassing simple grids, tables and trees, it also includes all kinds of treemaps involving the placement of rectangles. For instance, Slice and dice, Squarified, Strip and Pivot layouts are various points in this five dimensional space. Many classic statistics visualizations, such as 100% stacked bar charts, mosaic plots and dimensional stacking, are also instances of this class. A few new and potentially interesting points in this space are introduced, such as spiral treemaps and variations on the strip layout. The core algorithm is implemented as a JavaScript prototype that can be used as a layout component in a variety of InfoViz toolkits.

Index Terms—Layout, Visualization Models, Tables & Tree layouts, Grids, Treemaps (slice and dice, strip, squarified and pivot variations), Mosaic plots, Dimensional stacking.

1 Introduction

Treemaps are now over twenty years old [15]. This visualization technique has generated much enthusiasm in the information visualization community and has become a small research area of its own [22]. In the general public, treemaps have had their moments of fame with the map of the market [24], and are slowly making their way as a standard device in the toolkit of graphic designers [4, 25].

Yet, for a technology about as old as the World Wide Web, which keeps a high level of interest in the research community, we could hope for more salient success: treemaps and related rectangular space-filling approaches are still not a common visualization device of everyday use. Success stories for treemaps are the result of talented graphic design work, where the visualization designer has crafted the layout and visualization parameters to match a specific context and narrative.

We attribute the need for careful crafting to the lack of a good understanding of their design space. Choosing the right layout parameters to suit a particular dataset and features to be highlighted requires a solid experience and a dedicated presentation effort. In analysis contexts, this presentation effort is a distraction from the research task and therefore often sub-optimal layouts are used. This lack of presentation automation, or easier customizability, creates a barrier to more widespread adoption in the contexts where rectangular space-filling layouts, such as treemaps, could really bring insight.

Our goal here is to define more precisely the design space of a particular class of layout algorithms that lie at the root of the treemap concept: rectangular space-filling layouts, i.e. the layouts that tile a unit square with rectangles in a space-filling manner. We describe how input data is transformed into a set of rectangles that tile the unit square through a process that is constrained by the dimensions that span this design space. In addition we contribute a universal algorithm for some class of rectangular space-filling layouts. This universal algorithm is parametrized by functors that represent the described dimensions. We present the algorithm with various examples of useful values for these dimensions, which allow creating well known as well as novel rectangular, space-filling layouts. We believe that a solid understanding of this design space can serve as a basis to develop methods and heuristics that determine the most appropriate layout given a particular data set. Finally, it has been suggested that the design-space of space-filling rectangular layouts is very large and that the generic problem of creating such layouts fall in the category of NP-hard problems [9]. To the contrary, we show here that:

1. Useful rectangular space-filling layouts belong to a class of lim-

- *Thomas Baudel is with IBM ILOG Advanced Studies, E-mail: baudelth@fr.ibm.com.*
- *Bertjan Broeksema is with IBM ILOG Advanced Studies,Institute Johann Bernoulli, Univ. of Groningen, The Netherlands and INRIA, Bordeaux, France. E-mail:bertjan.broeksema@fr.ibm.com.*

Manuscript received 31 March 2011; accepted 1 August 2011; posted online 23 October 2011; mailed on 14 October 2011.
For information on obtaining reprints of this article, please send email to: tvcg@computer.org.

ited complexity, which we call sequential space-filling layouts. They have an average complexity of $O(n)$ or $O(nlogn)$, with worst case at $O(n^2)$.
2. Only five dimensions suffice to characterize the task of sequentially laying out a data set in space-filling rectangles: order, size, chunk, recurse and phrase (the chunk and phrase dimensions echo Buxton's perspective on the structure of input [10]).
3. Functional representations (functors) of these five dimensions are the parameters of a universal algorithm for the class of sequential, rectangular, space-filling layouts.

To support those assertions, we first state the problem more formally by specifying the class of problems we address and place our work in context. Next we describe the five dimensions of the design space spanning the introduced problem class. This is followed by the construction of an universal algorithm which covers this space. Next, we show how these dimensions serve as parameters of the algorithm and demonstrate how various values for each dimension result in different well-known or new layouts. Finally, we extend the algorithm to allow handling hierarchical data structures for the realization of a generalized treemap layout framework. This lets us conclude on the potential of our algorithm to help mastering better the design space of treemaps and rectangular space-filling layouts, as well as address more general visualization techniques with similar algorithmic space characterizations.

2 PROBLEM STATEMENT

There are various ways to define the concept of rectangular space-filling layouts which underlie the design space of treemaps. We choose to state the problem in a more general, yet simple to formalize way. The problem we address can be stated formally as follows. A *rectangular space filling layout* is an algorithm which

- taking as input an ordered list of N positive integers: $\{a,b,c,...\}$, whose sum is equal to S,
- outputs a tiling of the unit square $[0,1]\times[0,1]$ with N non-overlapping, rectangles of surfaces $\{a/S,b/S,c/S,...\}$. The sides of these rectangles are only allowed to be aligned with the sides of the unit square, i.e. no rotation is allowed. Colloquially, we call these "orthogonal" or "manhattan" rectangles;
- while maximizing a given objective function;

The objective function will define, for a large part, the algorithm to apply. This function is generally a weighted sum of objectives [17] involving criteria such as:

1. Aspect ratio of rectangles should be close to 1 or some chosen value.
2. Preservation of the input order.
3. Stability to resizing or adding or removing a small number of items.

Other criteria could be interesting to investigate, such as including the input surfaces as part of the objective function instead of posing them as hard constraints.

While the layout phase is the center of our attention, treemaps are often perceived as a method to visualize hierarchies. To address this perception, our generalized approach applies the solution for the above layout problem to each level of the hierarchy.

This definition discards some interesting work from our focus: Ellimaps [20] or Voronoi Treemaps [1] for instance, are moved out of our scope. Still, it covers most of the central area of treemap research: slice and dice, squarified, pivot layouts. We stress, however, that our approach is not limited to treemaps but also covers related rectangular space-filling visualizations such as mosaic displays, icicle plots and 100% stacked bar charts.

2.1 A Generic Algorithm

We propose a generic algorithm, whose parameters allow specifying points (i.e. concrete rectangular, space-filling layouts) in a specific portion of the problem space described above. For convenience, in the remainder of the paper we will use the following conventions. Our algorithm takes as input a collection of N tuples [14], which we will denote T. We refer to a single element in T as $t_i \in T$ with $0 <= i < N$. Each tuple t_i has a finite number of attributes, which we denote a_j for the jth attribute.

The output of our algorithm is a representation of the input data presented as a list of graphic instructions in the form $drawRectangle(t_i,x_i,y_i,w_i,h_i)$ for each i. This list is constrained: output rectangles do not overlap and together they fill the unit square. In our algorithm description, we consider that the output is produced through a succession of calls to a rendering function: $render : T \times Rectangle \rightarrow Graphical\ Output$. Hence, our algorithm can be characterized as a function that takes a collection of tuples T and a renderer R: $draw : T \times R \rightarrow Graphical\ Output$.

2.2 Sequential methods

Considering that our target problem is defined as a combinatorial optimization problem, it would seem, at first sight, that reaching optimal solutions is a hard problem. Some have assumed that the disjunctive nature of the problem makes it NP-hard [9]. We do not adhere to this view. We will show that the constraints for tiling the full surface impose some severe restrictions on the allowed choices, and that dynamic-programming-based solutions work satisfactorily.

However, it remains clear that describing the full algorithmic design space of rectangular space-filling layouts, of unbounded complexity, is a challenging task. Rather, we decide to focus on a class of greedy methods, which we call *sequential layouts*. These methods are not allowed to use backtracking techniques of unbounded depth. They can perform a fixed number of passes on the input set and partition the input set to apply to each partition element a further layout method (divide and conquer/dynamic programming approach). This class of algorithms corresponds to the class of input-linear visualizations [2, 3], augmented with partial recursion capability. As defined in [2], algorithms are input-linear (or data-linear) when there is a constant K such that, for any input set T of length N, for all $i < N$, t_i is accessed at most K times. Augmented with local recursion, the algorithms we consider are contained in the class that we call $quasi-input-linear$ algorithms, defined by the proposition that t_i is accessed $log(N)*K$ times on average and $N*K$ at most, which gives algorithms in this class a worst-case complexity of $O(N^2)$.

There are several reasons to restrict oneself to exploring this class:

- All widely known algorithms that tile rectangles belong to this class.
- If further improvements are needed, local optimization techniques provide a range of simple techniques to move down to local minima.
- A design space of this class of algorithms can be fully characterized very simply.

Algorithms in this class, are conveniently written as a sequence of ordered passes (sequences) over the dataset, which is why we call this class "sequential algorithms" (or methods').

3 RELATED WORK

In the previous section, we have restricted our ambitions to provide for a simpler model. Still the space-filling model we describe covers a large number of commonplace views which are used in various contexts.

Tables, grids (figure 1d), file browsers, but also simple hierarchical trees are representable in our model. For the later case, it suffice to replace the rectangle renderer with a link renderer to obtain commonplace trees out of a space filling layout algorithm.

Rectangular space-filling layouts also cover a large portion of common statistical graphics. Equal height and 100% Stacked bar charts

Dimension	Signature	Description
Order	$T \to T$	order in which the data items are laid out.
Size	$T \to \mathbb{R}$	how individual data items are sized.
Chunk	$C \times \mathbb{R} \to \mathbb{B}$	how data items are chunked together.
Recurse	$C \to \mathbb{B}$	how a chunk lays out its content.
Phrase	$C \to (Side, Direction)$	how chunks are assembled in the available space.

Table 1: Overview of the five dimensions that span the design space of sequential, rectangular space filling layouts. All functors are stateful, meaning their actual signature is $X \times state \to Y \times state$

are two of the simplest space-filing layout to implement. More interestingly, treemaps are predated by Mosaic plots [12, 13], which are hierarchical, space-filling layouts too. Dimensional stacking [18] and pixel bar charts [16] are also covered by our design space.

Starting from the seminal paper of Johnson and Shneiderman [15], treemaps have been a widely explored area. The issue of the poor aspect ratios produced by the slice and dice technique have been addressed quite rapidly after: M. Hascoet-Zizi proposed, as soon as 1992, the squarified layout algorithm [31]. Bruls et al. [9] rediscovered this technique and popularized it. Around the same time, M. Wattenberg proposed a variety of pivot based layouts, while B. Bederson introduced strip and quantum layouts [5]. A recent technique worthy of discussion is spatially ordered treemaps [30]. We refer to Shneiderman [22] for a more exhaustive listing. All the techniques presented above are particular instances of our generalized layout technique.

Of interest to our work are some recent efforts to model the treemap design space more formally. Onak et al. introduced "Fat Polygonal Partitions" [6] and "Circular partitions" [19] which are interesting variants of the problem we address, splitting the unit square with quadrilaterals instead of rectangles. The extra degree of freedom gained lets them improve the optimization results, and in particular provide better overall aspect ratios. Interestingly enough, both techniques can be recreated with a generalization of our universal layout algorithm wherein the output is changed to general quadrilaterals. Schulz et al. [21] survey the design space of implicit hierarchical visualization. They define layout as one of the axis of this space. However, they only distinguish between subdivision and packing, which leaves layout still as an ill-defined black box. We show that layout can be defined more precisely and covers a design space on its own. Vliegen et. al. [27] discuss several of the dimensions we consider for our space (direction, size, nesting as a equivalent to our partitioning operator, but not our recurse functor), as well as other dimensions (transformation, uniform density), but those dimensions are treated as parameters of finite range (or black-box parameters) and the algorithm features that are the foundation of the problem space are not discussed.

Our work inscribes itself in a variety of foundational work on describing the structure of graphical representations. Our approach is inspired by some seminal work in formalizing the design space of information visualization such as Chi et al. [11]. Bertin's seminal Semiology of Graphics [7] provides numerous insights, including a treemap layout (p. 270). Wilkinson's Grammar of Graphics [29] went much further by including data projection and statistical concerns in his framework. This work was extended by Wickham and Hofmann [28] who propose 1d and 2d primitives for map data to various space-filling and non-space-filling plots. But we are not aware of related work regarding decomposing and rationalizing the layout problem. We propose to characterize layout classes as a space whose dimensions are characterized by elementary decision functions. We open such a possibility by restraining our taxonomy to a useful subset of algorithmic classes, that happens to be easy to study and categorize: input-linear and quasi-input-linear algorithms. Slingsby et al. [23] proposed a general technique to configuring hierarchical space-filling layouts. Our design space is structured quite similarly, but has been inspired by our Discovery earlier work [3]. Our contribution over the work of Slingsby et al. is that, instead of providing a preset number of layout functions, we request the specification of five independent parameters: order, size, chunk, recurse and phrase. All are functional expressions determining the behavior of the sequential layout algorithm. These expressions allow covering the full design space of sequential space-filling layouts. In short, our model does not rely on a preset number of "black boxes" predefined layouts. All possible layouts in the class we consider are described through parameter settings only. Order and Size are similar to Slingsby et al. *sSize* and *sOrder* operator.s Slingsby's model can probably replace its *oLayout* operator with *oChunk*, *oRecurse* and *oPhrase* operators and make all those operators functional to reach the full expressiveness of our model.

4 Design Space

The design space for sequential, rectangular space-filling layouts is determined by five independent dimensions: *order*, *size*, *chunk*, *recurse* and *phrase*. Those dimensions are functional: they are functors that define choice points to be taken based either on the input data or on the current state of the layout process. For example, a particular instance of the *chunk* functor can make the decision to start a new chunk every tenth item. All functors are stateful, meaning that their actual signature is $X \times State \to Y \times State$ A brief overview of those functors is given in table 1.

4.1 Order

As the data is handled sequentially, the first dimension that determines the layout is the order in which to process input. The functor that defines this dimension takes as input the original data items to be laid out and return the items in a particular order. Basic ordering functions include sorting data items ascending or descending based on a particular input attribute a_i. More elaborate ordering functions can be used: for instance, returning all items at even indices followed by items at odd indices. Functionally this dimension is represented as:

$$order : T \to T$$

We must take in account an important restriction: because we cover *only* the class of sequential algorithms, only ordering functions that are sequential can be used in our model. This includes linear time sort algorithms such as bucket sort and divide and conquer sort algorithms such as quicksort.

4.2 Size

We have defined the layout problem as the tiling of a unit square, given a list of sizes. The size determines the relative space a given data item will take in the final layout. Retrieving these sizes from the original data can be expressed as a function on the input data. The most trivial functions return either a fixed size or use a particular numeric data attribute. Other possibilities include calculations on attributes of the data items or transforming categorical data attributes to a numeric value that serves as size based. Functionally this dimension is represented as:

$$size : T \to \mathbb{R}$$

Where, $size(T[i])$ returns the size of the ith tuple (i.e. L[i], from the initial problem statement).

4.3 Chunk

A *chunk* is a rectangle in the unit square that fully contains one or more of the output items. In the remainder, C denotes the list of chunks that pave unit square and c_i the ith chunk, with $0 < i <= N$. When $i \equiv N$, all items are laid out in separate chunks.

A consequence of the limitation that the algorithm must be sequential is that items must be placed sequentially in successive chunks. How big these chunks are, i.e. how many items are laid out at once, can be determined in various ways. Therefore, *Chunking* defines yet another dimension of our design space. The decision of adding an item to the current chunk or to the next can be based on various variables such as the number of items in a block with respect to the total number of items to layout or the aspect ratio of items in a chunk. Functionally this dimension is represented as:

$$chunk : C \times \mathbb{R} \rightarrow \mathbb{B}$$

Where, $chunk(c[j], L[i])$ returns false when $L[i]$ should be added to the jth chunk and true when a new chunk should be formed.

4.4 Recurse

The recurse dimension indicates whether, after having isolated a chunk, the algorithm should recurse into the chunk, reapplying itself to further improve the aspect ratio or other optimization goals. This is a partial recursion and it is bounded by the number of data items to be laid out. This functor therefore causes the algorithm's worst case complexity to grow into $O(N^2)$, just under the same kind of scenario as the quicksort algorithm: the worst case occurs when $N-1$ items are laid out in a chunk and in the recursion proceeds $N-2$ times, until finally no items are left to process. This results in $N(N+1)/2 = O(N^2)$ steps.

This dimension allows implementing the various types of pivot layouts. Recursion can be used for instance to improve the aspect ratio of small items in a strip treemap without compromising the ordering (1a). Functionally this dimension is represented as:

$$recurse : C \rightarrow \mathbb{B}$$

Where, $recurse(C[j])$ returns true when the algorithm must recurse into the jth chunk and apply itself recursively to the items in this chunk and false otherwise.

4.5 Phrase

When a chunk is finished it must be laid out in the available space. Once a chunk is laid out, no changes to the aspect ratio or to the location of the chunk can be made (because of fixed depth backtracking). Consequently, the way a chunk can be located in the available space rectangle is strongly constrained, as explained in figure 5: if a chunk was allowed to be placed anywhere in the available space, using any aspect ratio, it would be easy to create input sequences that defeat the space filling constraint and therefore the sequential constraint. Hence, there are only four possible locations for a new chunk: the four sides of the containing rectangle. Besides newly created chunks *must* take either full height (resp. width) and grow horizontally (resp. vertically) as items are added. Finally, items can be stacked in various directions inside a chunk. In horizontal chunks they can be either stacked from left to right or vice versa. In vertical chunks they can be stacked from top to bottom or vice versa. This brings the total number of possible block configurations to eight. For completeness we note that items could be stacked vertical in horizontal chunks and stacked horizontal in vertical chunks, doubling the number of configurations. Functionally this dimension is thus represented as:

$$phrase : C \rightarrow (Side, Direction)$$

With $Side$ in $\{North, South, East, West\}$ and $Direction$ in $\{Up, Down, Left, Right\}$. Here, $phrase(C[j])$ returns the layout configuration for the jth chunk.

5 Algorithm

Now that we have specified the dimensions that span the design space of rectangular, sequential space-filling layouts, we can stitch them together in order to construct a universal algorithm that creates rectangular space-filling layouts in a sequential manner. Figure 2 summarizes the steps required for creating rectangular space-filling layouts.

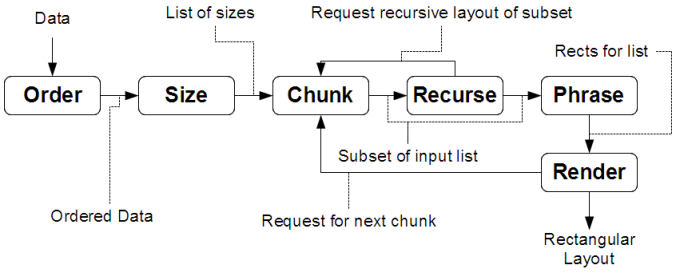


Fig. 2: Functional view of the algorithm, showing dependencies among components.

It gives an functional view of the algorithm and shows the various functors that transform data into rectangles.

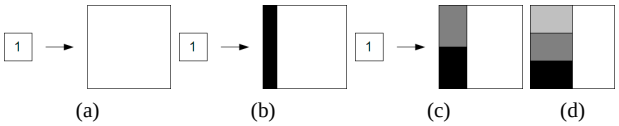


Fig. 3: Stacking items in a chunk.

5.1 Chunks

Our algorithm stacks items in a chunk as shown in figure 3. The chunk is placed on the left side of the unit square, and three out of six equally sized items are added to the chunk. Items are stacked from bottom to top. The order of stacking is depicted by color, going from black for the first to light gray for the last item. In figure 3a, no item is added, hence the chunk height is equal to the available space plane and its width is zero. Next, in figure 3b one item is added and the chunk now has a width proportional to the size of the item that was added previously over the total size to be laid out. In figure 3c a second item is added. The width of the chunk again grows proportionally, but the items' height are reduced as they are stacked in the chunk. Finally, figure 3d shows the result of adding yet another item to the chunk.

A chunk has six properties: *side*, *direction*, *fromX*, *toX*, *fromY* and *toY*. These are initialized using an initial chunk configuration (combination of side and direction), and the available space rectangle, but with a flat rectangle along the progress direction. Once no more items are to be added to a chunk, it can reduce the available space rectangle by moving one of its corresponding borders proportionally to the chunk's surface. Assuming that figure 3d shows the final state of the chunk, the available space would be reduced to $[x : 0.5, y : 0, w : 0.5, h : 1]$. This progressive reduction of the available space is materialized by the *reduce* function in our algorithm.

5.2 Chunking

Chunking is the process of deciding whether t_i gets added to c_j or c_{j+1}. This decision is based on a simple scoring function which takes a chunk c_j, the size of tuple t_i and returns a score s. Or more formally:

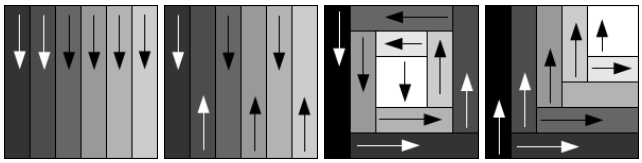
$$score : C \times \mathbb{R} \rightarrow \mathbb{R}$$

with $score(c_j, t_i) >= score(c_j, t_{i+1})$ when adding item t_{i+1} is considered to be an improvement of the layout. Like the main functors of our algorithm, *score* is stateful as well. The function describes an optimization function whose successive local maxima are used as chunk delimiters as the algorithm progresses through the input. This chunking process is described as pseudo code in listing 1.

```
1 function chunk(Array T) {
2   // size and score are global function pointers
3   Chunk currentChunk = new Chunk();
4   Chunk chunks[] = [currentChunk];
5   float prevScore = -inf;
6   for (var i = 0; i < N; ++i) {
```

```
  float curScore = score(currentChunk, size(T[i]))
  if (curScore < prevScore) {
    currentChunk = new Chunk();
    chunks.append(currentChunk);
    prevScore = score(currentChunk, size(T[i]));
  } else {
    prevScore = curScore;
  }
  currentChunk.add(T[i]);
 }
 return chunks;
}
```

Listing 1: The chunking algorithm



(a) (b) (c) (d)

Fig. 4: Four data independent phrasing strategies to create a space filling layout: (a) Strip; (b) Zigzag; (c) Spiral; and (d) Spikes.

5.3 Phrasing

When a new chunk is started, it must be decided how the chunk will be laid out in the available space. Recall that a chunk must be placed along one of the four sides of the available space, and that a stack direction must be given as well. Those two characteristics, taken together, form a chunk configuration. *Phrasing* is the process of picking a configuration for each successive chunk. Phrase is a functor that takes the previous chunk c_{i-1} and returns the chunk configuration for chunk c_i. That is, $phrase : chunk_j \rightarrow (side, direction)$. The first chunk is phrased according to the initial configuration set given to the algorithm, which is one of the above mentioned 16 possibilities. Depending on the strategy, the next configuration can be determined in a number of ways. There are four simple and useful strategies, which we call data independent. An overview of these strategies is shown in figure 4. Color depicts order of placement from first (dark) to last (light) and the arrow depicts the direction in which the items are placed in the chunk. These strategies work as follows:

- Strip (fig. 4a) - Using this strategy, each chunk is put the same way in the available space. This results in each first item of c_i being placed next to the first item of c_{i-1} for $i > 0$, i.e. a discontinuous placement of items.
- Zigzag (more accurately, Boustrophedon) (fig. 4b) - Is similar to strip except that the stack order of the items is reversed for every new chunk. Each first item of c_i is placed next to the last item of c_{i-1} for $i > 0$, i.e a continuous placement of items.
- Spiral (fig. 4c) - In this strategy, chunks are laid out against the next border of the available space based on the border and layout direction of the previous chunk. Each first item of c_i is placed next to the last item of c_{i-1} for $i > 0$, i.e a continuous placement of items.
- Spikes (fig. 4d) - Chunks are laid out perpendicularly to the previous chunk so that the first item of c_i is close to the first item of c_{i-1} for $i > 0$, i.e. a discontinuous placement of items.

5.4 Layout

All the basic components of our layout algorithm have now been introduced. The actual layout algorithm can be described by extending the earlier presented *chunk* function as follows. The first chunk is given an initial configuration. Next we start chunking as detailed before. However, before a new chunk is started, we first test if recursive layout is required and apply the algorithm recursively if so. Next, the phrase functor is used to determine the configuration for the new chunk. The resulting pseudo code is listed in listing 2.

```
function layout(Array T, int from, int to) {
 // order, size, score recurse and phrase
 // are global function pointers.
 Rect availableSpace = new Rect(0,0,1,1);
 // phrase(null) returns a default configuration
 Chunk currentChunk = new Chunk(phrase(null), availableSpace);
 int currentFrom = from;
 Chunk result[] = [currentChunk];
 T = order(T, from, to);
 float prevScore = -inf;

 for (i = from; i < to; ++i) {
  float itemSize = size(T[i]);
  float curScore = score(currentChunk, itemSize);
  if (curScore < prevScore) {
   // reduce is described in section 5.1
   currentChunk.reduce(availableSpace);

   if (recurse(currentChunk))
    Chunk recursiveChunks[] = layout(T, currentFrom, to);
    if (!recursiveChunks.isEmpty())
     result.pop(); // replace last chunk with
                   // subdivision result.
     result.append(recursiveChunks);

   currentChunk = new Chunk(phrase(currentChunk),
                            availableSpace);
   result.append(currentChunk);
   currentFrom = i;
   prevScore = score(currentChunk, itemSize);
  } else {
   prevScore = curScore;
  }
  currentChunk.addItem(itemSize);
 }
 if (currentFrom != from && recurse(currentChunk)) {
  Chunk recursiveChunks[] = layout(T, currentFrom, to);
  if (!recursiveChunks.isEmpty())
   result.pop(); // replace last chunk with subdivision result.
   result.append(recursiveChunks);
 }
 return result;
}
```

Listing 2: Basic layout algorithm

Finally, we can implement the draw function mentioned in the input and output section as listed in 3. The *chunk.rectangle* method simply assigns iteratively a rectangle to each item in the chunk based on the chunk's rectangle and its *direction* attribute and increments a local counter to assign the border of the next item to be drawn.

```
function draw(T, R) {
 Chunk chunks[] = layout(T, 0, T.length());
 foreach (chunk : chunks) {
  R.drawRect(chunk.rectangle());
  foreach (t : chunk) { // item in chunk
   R.drawRect(t, chunk.rectangle(t));
  }
 }
}
```

Listing 3: Draw function implementation

This fully describes our universal algorithm to solves the problem stated in section 2 in with sequential methods. Note that some parts of the algorithm are optimized in practice. For example, the *ordering* and *sum* functors are memoized, instead of being recalculated for each recursive iteration. These kind of optimizations are left out for clarity.

5.5 Completeness

As we have mentioned in the introduction, the dimensions we describe are independent, and each cover a functional, recursively enumerable, domain. We can show in a nonconstructive way that our generic algorithm allows the depiction of *any* sequential, rectangular, space-filling layout as defined in 2.2: the *order* functor is instantiated with a general function that can perform *any* calculation and store its result in

the state variable. The *chunk*, *phrase* and *recurse* functors are then free to reuse those results as they see fit. Hence it is always possible to ”pack” a knowingly sequential algorithm into the *order* functor, which has the adequate signature, and leave the other functors just retrieve precomputed values to perform the layout. Finally, as long as the *order* function stays sequential in the sense of 2.2 and the other functions stay in constant time, all possible parametrizations of our generic algorithm stay sequential.

While this sketched proof somewhat downplays the use of separate *phrase* and *chunk* functors, these two dimensions greatly simplify the expression of a large variety of rectangular space-filling layouts, because they allow expressing in a terse way the base ingredients of a divide and conquer approach to layout, which is a most sensible technique to encompass efficiently the various objective functions of the problem stated in 2.

Indeed, the combination of *chunk* and *phrase* functors capture the fact that the chunking process needs to proceed by growing chunks along the borders of the area to fill. If a sequential layout algorithm decided to attribute to a chunk a rectangle that is *not* on the border of the available area, and does *not* extend to the full height or width of this area, then it suffice to input the algorithm two (or more) additional items which cannot be made to fit the remaining space: the algorithm fails to find a proper tiling. Figure 5 illustrates this fact. In other words, an algorithm that wishes to chunk some items together into a distinctive block to further layout this block independently *must* proceed by placing the distinctive block along one of the edges of the area to layout, *or* be allowed to perform backtracks of arbitrary depth to decide how to organize those blocks together in the alloted space. These backtracks of arbitrary depth push the complexity of the method beyond the realm of input-linear and quasi-input-linear methods.

These considerations let us claim that our algorithm is universal for the class of algorithms considered, i.e. sequential methods for tiling the unit square with rectangles of varying surfaces. Still, we acknowledge a detailed proof is desirable. Such a proof goes beyond the scope of this paper as it would require further formalization and introduction to the class of input-linear algorithms. Instead, we now show how in practice those five dimensions can be combined to provide a wide variety of layouts.

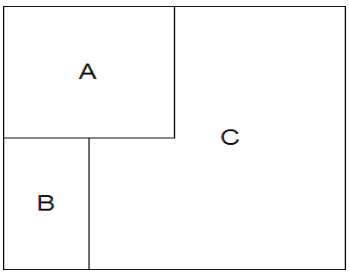


Fig. 5: When it is decided to place chunk *A* of size 1 in this square of surface 4; and if there are only two items left to fit in the square which sizes are 0.5 (B) and 2.5 (C), then there is no way to produce a rectangular space-filling tiling of the square.

6 Layout Parameters

This section presents various examples of functions for each of the considered dimension and demonstrates how different configurations of the algorithm creates instances of well known or new layouts.

First, we briefly discuss the *order* and *size* functions as these are rather trivial. Next, we illustrate how the *score* and *phrase* functions interplay with each other to allow producing a variety of known or novel layouts. Finally, we discuss the recursion functor which adds the ability to reenter the layout algorithm inside chunks.

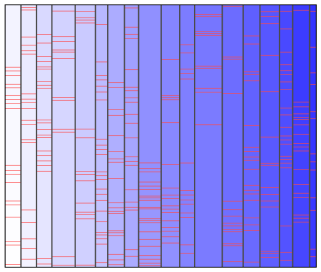
In the following we use a dataset containing US cities, the states they belong to and measures for various properties such as population, climate, education and health care. Individual items are colored by item index from light (low index) to dark (high index) and separated by red lines. Chunks are separated by black lines.

6.1 Order and Size

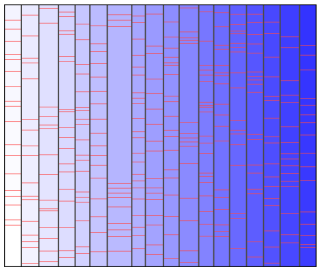
The *order* functor orders (a subset of) the original input data set. When this functor is not specified the original order of the data is kept. A trivial example of an ordering function is one which sorts the tuples based on a particular attribute of the data set, e.g. by population. However, we explicitly named this dimension *order* as more complex ordering, which falls outside the range of sorting are allowed. For example, we chose to access first the tuples that have an even index, followed by the tuples that have an odd index.

The *size* functor lets one specify which attribute or function to use to compute the size of each tuple. There are only a limited number of useful *size* functions:

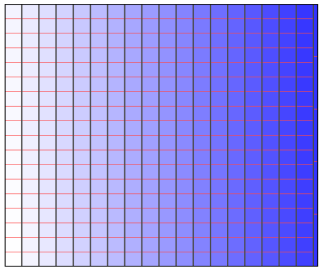
- An function that returns a tuple attribute (fig. 6a) which should be a ratio attribute for best result. For nodes in a hierarchy, the value of this function is the sum of all values of the tuples it contains. An efficient implementation will of course cache this sum for each node to avoid recomputing it needlessly.
- The result of a computation on tuple attributes (fig. 6b). This is merely a special case of the former where a value is calculated based on one or more fields of the available data.
- Constant (fig. 6c). One for tuples and nodes alike. This is useful for similar usages in grid layouts.



(a) Population



(b) Education over HealthCare



(c) Constant

Fig. 6: Various settings for the size functor.

6.2 Chunk scoring

The Chunk scoring function is the most powerful and complex parameter. The *score* functor evaluates a score for a chunk composed of a current chunk and one extra item. So, this *score* functor is evaluated *before* adding an item to the chunk. In the remainder of this section we assume a global variable state is available for bookkeeping, which is updated by the algorithm as required.

The simplest instances of this functor are *Slice* and *Dice*, which return the maximal score when the chunk has only one item, resp. when its cardinality is the full node. When these methods are alternatively applied at each level of a hierarchical data structure, this produces the regular slice and dice layout. Equally simple is the *Grid* function. This

function returns its maximum when the number of items is equal to the closest approximation of the square root of the number of children in the node. This results (approximately) in a number of chunks with equal cardinality (To produce perfect grid, the *size* functor needs to be adjusted to return the smallest square integer above the cardinal of the input size). As its name indicate, this chunking method produces grids (fig. 6c) if the size attribute is set to constant, or some variant of a strip layout if not.

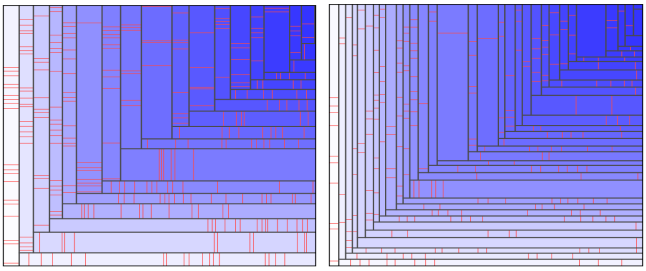
```
function Slice(Chunk c, double itemSize, State state) {
  return c.itemCount == state.itemCount ? 1 : 0;
}
function Dice(Chunk c, double itemSize, State state) {
  return 1 - c.itemCount;
}
function Grid(Chunk c, double itemSize, State state) {
  return square(c.itemCount) <= state.itemCount ? 1 : 0;
}
```

Listing 4: Elementary chunking functions

More interesting, yet complex, chunking functions can be used, allowing for instance to implement a variety of squarified and strip layouts. The *bestAverageAspectRatio* (lst. 5, fig. 7a) and *bestMinAspectRatio* (list. 6, fig. 7b) functions return as a score respectively the average aspect ratio of all the items in the chunk and the aspect ratio of the smallest item in the chunk. The closer this value is to one, the more "fit" the current chunk is to be laid out. To compute these values, the width (resp. height) of the chunk has to be computed by evaluating the ratio of the sum of the current chunk over the total sum of the square to be laid out. The height (resp. width) is equal to that of the enclosing space. Having computed the width and height of the chunk makes computing the average and minimal aspect ratios of its content elementary.

```
function BestAverageAspectRatio(Chunk c, double itemSize, State state)
    {
  Rect rect = state.availableSpace;
  float expandingDim = c.isVertical ? rect.height : rect.width;
  float fixedDim = c.isVertical ? rect.width : rect.height;
  int newChunkSize = c.sum() + itemSize;
  float aspectRatio = expandingDim / (c.itemCount + 1)
    / (fixedDim * newChunkSize / state.overallSize);
  return aspectRatio > 1 ? 1 / aspectRatio : aspectRatio;
}
```

Listing 5: Calculating score based on average aspect ratio



(a) Best Average (b) Best Minimum

Fig. 7: Two scoring functions based on aspect ratio of items: (a) best average aspect ratio. (b) best aspect ratio for smallest item. The *order* function is not specified, resulting in a (intentional) poor squarification of the layout.

```
function BestMinAspectRatio(Chunk c, double itemSize, State state) {
  Rect rect = state.availableSpace;
  float expandingDim = c.isVertical ? rect.height : rect.width;
  float fixedDim = c.isVertical ? rect.width : rect.height;
  int newChunkSize = c.sum() + itemSize;
  int minItemSize = c.minItemSize() < itemSize
    ? c.minItemSize() : itemSize;
  float aspectRatio = expandingDim * (minItemSize / newChunkSize)
    / (fixedDim * newChunkSize / state.overallSum);
  return aspectRatio > 1 ? 1 / aspectRatio : aspectRatio;
}
```

Listing 6: Calculating score based on aspect ratio of smallest item

Finally, to implement Pivot layout, other *score* functions can be used that split the input data in two approximately equal parts. Pivot by middle has its maximum around the middle of the item count, Pivot by size has its maximum at the index of the biggest item, and Pivot by split size has its maximum where the sum of the sizes are closest to half the total sum.

```
function PivotByMiddle(Chunk c, double itemSize, State state) {
  return -square(c.itemCount + 1
              - (state.to - state.from) / 2);
}
function PivotBySplitSize(Chunk c, double itemSize, State state) {
  return -square(state.currentChunk.size() + itemSize
              - state.remainingSum / 2);
}
function PivotBySize(Chunk c, double itemSize, State state) {
  return state.currentIndex == state.indexOfBiggestItem ? 1 : 0;
}
```

Listing 7: Pivot layout functions: The first two are parabolas which have their maxima at the desired item index (Fig. 1c).

6.3 Phrase

In section 5.3 we introduced four simple, data independent phrasing strategies. The implementation of these strategies is done by means of functions which basically consist of a switch statement over the current chunk phrase configuration to pick the next one. A novel layout is shown in figure 1b, where we apply spiral phrasing to our US population dataset.

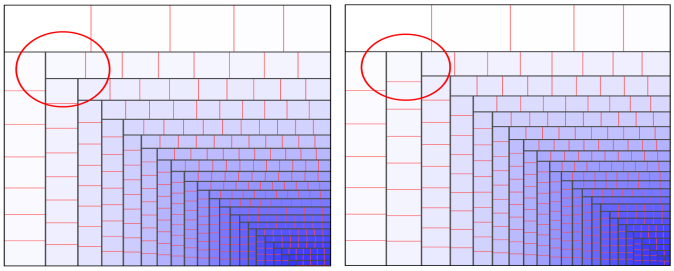
More complex phrasing functions, combined with recursion, can use some state variables and a stack to produce sophisticated layouts such as the Peano-Hilbert curve of figure 1d.

```
function hilbertPhrasing(state) {
  state.sequence += 1;
  var direction = state.parentConfig;
  if (state.depth % 2 === 0) {
    // at even depth, we split in 2 blocks of 2 and reverse
    // direction of the 2nd chunk
    if (state.sequence % 2 === 0) { return direction; }
    else { return direction.reverse(); }
  } else {
    // at odd depths we toggle the orientation of each quarter
    if (state.parentSequence % 2 === 0) { // 1st half
      if (state.sequence % 2 === 0) {
        return direction.alternate(); } // q1
      } else { return direction; } // q2
    } else { // 2nd half
      if (state.sequence % 2 === 0) {
        return direction.alternate(); } // q3
      } else { return direction.alternate().reverse(); } // q4
    }
  }
}
```

Listing 8: Hilbert phrasing (see fig. 1d). direction, sequence and depth are state variables maintained at the scope of each chunk.

In addition, phrasing can be made dependent on the input data, to produce data dependent strategies. Those can echo the data independent strategies described earlier. The data dependent alternative of Strip phrasing is called Worst Discontinuous phrasing. In this strategy, each chunk is placed so as to degrade the aspect ratio of the available space. Items in new chunks are stacked in the same direction, resulting in a discontinuity at each chunking step. The data dependent variant of zigzag is the Worst Continuous strategy which places the next chunk so as to degrade the aspect ratio and reverse the stack direction. The data dependent variant of Spiral is Best Continuous. In this strategy the chunk is placed so as to improve the aspect ratio of the available space and orders the items so that the first item of the next chunk is next to the last item of the previous chunk. Finally, the data dependent variant of Spikes (fig. 8a) is called Best Discontinuous (fig. 8b). It places

each chunk so as to improve the aspect ratio of the available space and so that the first item of the new chunk is next to the first item of the previous chunk. This layout is a novel and interesting improvement over the regular squarified layout as it results in significantly squarer items, as shown in figure 8b.

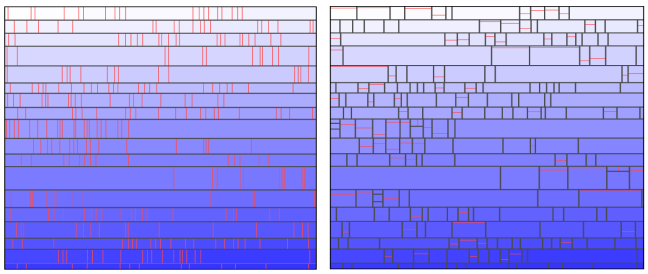


(a) Spikes (b) Best Discontinuous

Fig. 8: Data independent and dependent spike phrasing strategies. Notice the two vertical chunks at the left in (b).

6.4 Recurse

As with the other dimensions, *recurse* is a functor in our algorithm that returns a boolean. When true is returned the layout algorithm is applied to the items of the current chunk. If the recursion results in one or more chunks, the current chunk is removed from the result and replaced by the chunks that resulted from the recursion step. Obviously, there must be a stop criterion for the recursion. Therefore, the simplest value of the recurse functor should not just be a function that returns true but limit itself based on the number of remaining items. In our implementation it stops when only two or less items are left to be laid out.



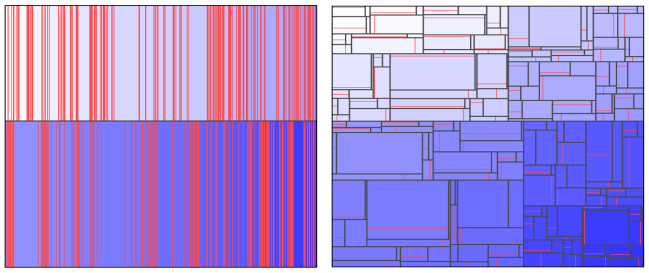
(a) Without recursion (b) With recursion

Fig. 9: Strip layout with recursion to improve small item visibility.

One reason to use recursion is that it might improve aspect ratio of small items. Figure 9 shows that after recursion many of the small items got a better aspect ratio. Another reason for recursion is to implement various variants of the well known pivot layouts. Figure 10 shows how recursion complements the pivot by middle scoring function.

7 Structuring

So far, we have focused purely on the layout aspect of space-filling visualizations. However, as stated before, treemaps are a central concern, and those are most often perceived as a method to visualize hierarchies. To accommodate this perception, we extend the presented algorithm by introducing a structuring phase. This phase consist of partitioning and ordering which reflect the *sHier* and *sOrder* states from Slingby et al. [23]. First we define a new structure Node, used for representing hierarchies. This allows the algorithm to take as input either a tabular input data set and turn it into an ordered hierarchy according to some set criteria, or a preexisting hierarchy of ordered



(a) Without recursion (b) With recursion

Fig. 10: Recursion to implement pivot by middle: (a) size by population, Strip, no recurse (b) size by population, Strip, recurse #item $>$ 2.

nodes. Each node has a unique identifier and keeps a list of children. The children can be tuples, nodes or a mix of both.

We adapt the previously introduced *layout* and *draw* functions to take this new context into account. First, the layout function now takes a node *n* as input and additionally does not assume the unit square as available space any longer but takes a bounding rectangle *BR*. Furthermore, the children of a node can be a mix of tuples and nodes. Hence, the size function must be adapted to return the appropriate size for the *ith* child of *n*. The required changes for the layout function are listed in 9.

```
1  layout(Node n, Rect BR, int from, int to)
2   Rect availableSpace = BR;
3   // Same as in listing 2
4   int N = n.childCount();
5   int S = sum(n, from, to).
6   order(n, from, to);
7   float prevScore = -inf
8   for (int i = from; i < to; ++i)
9    int itemSize = size(n.children[i])
10   float curScore = score(currentChunk, itemSize)
11   if (curScore < prevScore)
12    // Same as in listing 2
13   else
14    prevScore = curScore;
15   currentChunk.add(n.children[i])
16  // Same as in listing 2
```

Listing 9: Layout function for hierarchical structures

Like the layout function, the draw function now takes a node *n* and a bounding rectangle *BR* instead of assuming unit square. The items in *b* are now elements *e*, which are either tuples or nodes. In the former case, we call draw on the Renderer as before. In the latter case, we recursively call draw, but now we set the bounding rectangle to the rectangle that was laid out for this node. All children of the node will therefore be laid out not in the unit square but in the input bounding rectangle. The adapted version of the draw function is listed in 10.

```
1 draw(Node n, Renderer R, Rect BR)
2  Chunk chunks[] = layout(n, BR, 0, n.childCount());
3  foreach(chunk : chunks)
4   foreach(element : chunk)
5    if (e instanceof Tuple)
6     R.drawRect(e, b.rectangle(e))
7    else // e is a Node
8     // Recursively draw the next level in the tree.
9     draw(n, R, chunk.rectangle(e))
```

Listing 10: Recursive draw function for hierarchical structures

At this stage our space-filling layout algorithm is able to deal with both flat and hierarchical data. In both cases the initial step is to convert input T into a root node containing T as children. Now an additional structuring function is added before calling draw, which converts the flat root node into an ordered hierarchy. The structure function executes in two phases, partitioning and ordering. These two phases are detailed now.

7.1 Partition

Partitioning aggregates tuples in nodes according to a user provided expression. This expression is expected to return a unique sub node identifier for each tuple, or null if the tuple is to remain at the current level. Calling this functor on each child tuple results in creating one additional level to the hierarchy. Listing 11 shows how a partitioner can be used to create a hierarchical structure out of flat data. Using this partition expression, data can be partitioned as listed in 11.

```
structuring(Node n, int depth)
 Array result = [];
 Map partitionMap = {};
 foreach (e : n.children)
  Object id = partitionExpression(e, node, depth)
  if(id != null)
   child = partitions.get(id);
   if(child == null);
    child = new Node(id);
    structure(child, depth + 1)
    partitions.put(id, child)
    result.add(child)
   child.add(e) // Add element to node
  else
   result.add(e) // Add element as leaf
 order(result);
 n.children = result
```

Listing 11: Structuring

The simplest partitioning expression, *Enumeration*, returns the value of a nominal tuple attribute: $returnget(t, myPartitioningColumn)$. This partitioner allows, for instance, to group a set of US cities by states. Likewise, date and numeric partitioners can be defined to split according to values and fields held in a date or numeric attribute. A hierarchical partitioner can be defined as a list of single column partitioners, resulting in mosaic plot layouts. Hierarchical partitioners need to keep track of the depth of the current node, to invoke the proper level of partitioning. Finally, a path partitioner takes an attribute column that describes a path (directory + file name, or URL, or date in the form y/M/d h:m:s, for instance). At each depth level *n*, this partitioner will return the corresponding *n*th substring in the attribute value. This partitioner is used for instance to display a file hierarchy in a treemap.

7.2 Order

Once partitioning is done, the resulting list can be ordered. Because the children can be either nodes or tuples, the function used to determine order needs to take into account the possibility of having to compare a node against a tuple. Past this hurdle, defining a comparator is fairly simple. The most common comparator, like the *Enumeration* partitioner, uses a column as its sort criteria. To handle nodes, this comparator defines an aggregation value for nodes, which can be, for instance, the sum of all the tuple values, or their average, maximum or minimum. Alternatively, the comparator can decide to place all nodes before or after all tuples, and sort them according to their node id, their immediate child count or the number of tuples they contain. Each of those possibilities will determine different placement of the nodes at a given level, but will not affect the layout algorithm per se. Finally, when ordering is done at the structuring phase, it should be obvious that ordering can be removed from the layout phase.

8 Conclusion

We have defined the design space of sequential, rectangular, space-filling layouts. This space is covered by five independent, functional, dimensions, namely: order, size, chunk, recurse and phrase. In addition we presented a universal algorithm for sequential, rectangular, space-filling layout. Our method leverages the observation that optimization techniques for sequential layout methods are essentially of the divide and conquer type, requiring chunking the input into blocks that are further laid out separately. The sequential nature of the process imposes the chunking process to proceed from a full side of the available space to fill, and then proceed along one of four possible sides, along one of four possible directions. Giving the user the ability to specify the value of each of those dimensions at each step of the algorithm results in a universal method to describe this class of algorithm. Additionally we discussed an extension of the algorithm in order to make it suitable for hierarchical data as well.

Our Discovery [3] framework implements the majority of the presented algorithms and techniques, even though it was not formalized at the time. To provide broader access, we have implemented the generalized space-filling layout algorithm as a small, independent component. This component consists of only a few hundreds of lines of JavaScript code and follows closely the presented algorithms.

A future research direction is to extend the presented algorithms to other layout problems involving only data-linear or quasi-data-linear visualizations. We have mentioned the possibility of describing circular partitions [19] and [6] with the same techniques. Many tree and graph drawing algorithms that do not require global heuristics (where the position of a node depends possibly on all the other node positions) are also representable with our technique, simply replacing the drawing of rectangles with drawing of arcs joining the centers of the rectangles.

Yet, returning to the original motivation of our work. We have already shown (figure 8b and 1a) that subtle changes to the configuration of the algorithm result in improved layouts. Relying on the designer to think about those possible changes is unrealistic. Hence, our goal in defining this layout design space is to provide a more solid grounding to support analysts in their use of space-filling displays. We envision using this design space to develop a systematic method and a set of heuristics to (semi-) automatically choose a proper layout given some data sets, appropriate meta-data and contextual information.

Finally, our work presents some aspects that seem to be new in the area of computational geometry. As said before, we are not aware of related work regarding decomposing and rationalizing layout design spaces relying on algorithmic properties of the problem specification. Similar under-specified problems (whose objective function is a weighted sum of objectives, the weights being personal decisions) are abundant in the literature. By leveraging the cases where these problems are satisfactorily addressed by algorithms of low complexity (sequential, dynamic-programming...), we have found a way to characterize such algorithmic design spaces elegantly, through a limited set of functors that represent elementary decision functions that are to be made by the algorithm: in our context, "how to group items together? (chunk)", "in what order to fill the plane? (phrase)", "do we refine the groups? (recurse)". The study of input-linear and quasi-input linear algorithms in such contexts could bring new insights to these classes of problems.

Acknowledgments

The authors like to thank the following persons for their reviews and thoughtful comments: Alex Telea from university of Groningen, Guy Melançon from INRIA, Bordeaux and our colleagues from IBM Frank van Ham and Christophe Jolif.

References

[1] M. Balzer, O. Deussen, and C. Lewerentz. Voronoi treemaps for the visualization of software metrics. In *Proceedings of the 2005 ACM symposium on Software visualization - SoftVis '05*, pages 165—-172, New York, New York, USA, 2005. ACM Press.

[2] T. Baudel. Visualisations compactes: une approche declarative pour la visualisation d'information. In *Proceedings of the 14th French-speaking conference on Human-computer interaction (Conference Francophone sur l'Interaction Homme-Machine)*, IHM '02, pages 161–168, New York, NY, USA, 2002. ACM.

[3] T. Baudel. Browsing through an information visualization design space. In *CHI '04 extended abstracts on Human factors in computing systems*, CHI EA '04, pages 765–766, New York, NY, USA, 2004. ACM.

[4] BBC. Superpower: Visualising the internet. web site: http://news.bbc.co.uk/2/hi/technology/8562801.stm, 2010.

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

[6] M. D. Berg, K. Onak, and A. Sidiropoulos. Fat Polygonal Partitions. *Computational Geometry*, 1(773):1–25, 2010.
[7] J. Bertin. *Sémiologie graphique*. Mouton, Paris, 1967.
[8] R. Blanch and E. Lecolinet. Browsing zoomable treemaps: structure-aware multi-scale navigation techniques. *IEEE transactions on visualization and computer graphics*, 13(6):1248–53, 2007.
[9] M. Bruls, K. Huizing, and J. Van Wijk. Squarified treemaps. In *Proceedings of the joint Eurographics and IEEE TCVG Symposium on Visualization*, pages 33–42, 2000.
[10] W. Buxton. Chunking and Phrasing and the Design of Human-Computer Dialogues. In *Proceedings of the IFIP World Computer Congress*, pages 475–480. North Holland Publishers, 1986.
[11] E. H.-h. Chi and J. Riedl. An operator interaction framework for visualization systems. In *Proceedings of the 1998 IEEE Symposium on Information Visualization*, pages 63–70, Washington, DC, USA, 1998. IEEE Computer Society.
[12] M. Friendly. Extending Mosaic Displays : Marginal , Partial , and Conditional Views of Categorical Data. *Journal of Computational and Graphical Statistics*, 8:373—-395, 1999.
[13] M. Friendly. A Brief History of the Mosaic Display. *Journal of Computational and Graphical Statistics*, 11(1):89–107, Mar. 2002.
[14] J. Heer and M. Agrawala. Software design patterns for information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12:853–860, September 2006.
[15] B. Johnson and B. Shneiderman. Treemaps: a space-filling approach to the visualization of hierarchical information structures. In *Proceedings of the 2nd conference on Visualization '91*, pages 284– 291, San Diego, California, 1991. IEEE Computer Society Press.
[16] D. Keim, M. Hao, and U. Dayal. Hierarchical pixel bar charts. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):255–269, July 2002.
[17] N. Kong, J. Heer, and M. Agrawala. Perceptual guidelines for creating rectangular treemaps. *IEEE Information Visualization*, 2010. To appear.
[18] J. LeBlanc, M. O. Ward, and N. Wittels. Exploring N-dimensional databases. In *Proceedings of the 1st conference on Visualization '90*, pages 230—-237, San Francisco, California, 1990. IEEE Computer Society Press.
[19] K. Onak and A. Sidiropoulos. Circular partitions with applications to visualization and embeddings. In *Proceedings of the twenty-fourth annual symposium on Computational geometry - SCG '08*, pages 28—-37. ACM Press, 2008.
[20] B. Otjacques, M. Cornil, N. Monique, and F. Feltz. CGD — A New Algorithm to Optimize Space Occupation in Ellimaps. In T. Gross, J. Gulliksen, P. Kotzé, L. Oestreicher, P. Palanque, R. O. Prates, and M. Winckler, editors, *Proceedings of the 12th IFIP TC 13 International Conference on Human-Computer Interaction: Part II*, volume 5727 of *Lecture Notes in Computer Science*, pages 805—-818, Uppsala, Sweden, 2009. Springer-Verlag.
[21] H.-J. Schulz, S. Hadlak, and H. Schumann. The design space of implicit hierarchy visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 17(4):393–411, 2010.
[22] B. Shneiderman. Treemaps for space-constrained visualization of hierarchies. http://www.cs.umd.edu/hcil/treemap-history/, 2009.
[23] A. Slingsby, J. Dykes, and J. Wood. Configuring hierarchical layouts to address research questions. *IEEE transactions on visualization and computer graphics*, 15(6):977–84, 2009.
[24] SmartMoney.com. Map of the market. web site: http://smartmoney.com/marketmap, 1998.
[25] The New York Times. Health of the car, van, suv, and truck markets. web site: http://www.nytimes.com/ imagepages/2007/02/25/business/ 20070225_ CHRYSLER_ GRAPHIC.html, February 25, 2007.
[26] F. Vernier and L. Nigay. Modifiable treemaps containing variable-shaped units. *IEEE information Visualization*, 2000.
[27] R. Vliegen, J. J. van Wijk, and E.-J. van der Linden. Visualizing business data with generalized treemaps. *IEEE Trans. Vis. Comput. Graph.*, 12(5):789–796, 2006.
[28] H. Wickham and H. Hofmann. Product plots. *IEEE Trans. Vis. Comput. Graph.*, 17(12):2223–2230, 2011.
[29] L. Wilkinson. *The grammar of graphics*. Statistics and computing. Springer, 1999.
[30] J. Wood and J. Dykes. Spatially ordered treemaps. *IEEE transactions on visualization and computer graphics*, 14(6):1348–55, 2008.
[31] M. Zizi and M. Beaudouin-Lafon. Accessing hyperdocuments through interactive dynamic maps. *Proceedings of the 1994 ACM European conference on Hypermedia technology - ECHT '94*, pages 126–135, 1994.