Optimal BSPs and Rectilinear Cartograms

Mark de Berg Elena Mumford Bettina Speckmann
Department of Mathematics and Computer Science, TU Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

`mdberg@win.tue.nl, e.mumford@tue.nl, speckman@win.tue.nl`

Abstract

A cartogram is a thematic map that visualizes statistical data about a set of regions like countries, states or provinces. The size of a region in a cartogram corresponds to a particular geographic variable, for example, population. We present an algorithm for constructing rectilinear cartograms (each region is represented by a rectilinear polygon) with zero cartographic error and correct region adjacencies, and we test our algorithm on various data sets. It produces regions of very small complexity—in fact, most regions are rectangles—while still ensuring both exact areas and correct adjacencies for all regions.

Our algorithm uses a novel subroutine that is interesting in its own right, namely a polynomial-time algorithm for computing optimal binary space partitions (BSPs) for rectilinear maps. This algorithm works for a general class of optimality criteria, including size and depth. We use this generality in our application to computing cartograms, where we apply a dedicated cost function leading to BSPs amenable to the constructing of high-quality cartograms.

Categories and Subject Descriptors: F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems—*Geometrical problems and computations*

General Terms: Algorithms, Experimentation

Keywords: Geometric algorithms, indexing structures, binary space partitions, cartograms, automated cartography.

1 Introduction

Cartograms. A cartogram is a thematic map that visualizes statistical data about a set of regions like countries, states or provinces. The size of a region (measured in area) in a cartogram corresponds to a particular geographic variable, for example, population [5]. Since the sizes of the regions are not their true sizes they generally cannot keep their shape. Ideally—to preserve recognizability—the deformation should not change the topological structure of the map, that is, each region should keep the same neighbors.

Globally speaking, there are four types of cartograms. The standard type (the *contiguous area cartogram*) has deformed regions so that the desired sizes can be obtained and the adjacencies kept. Algorithms for such cartograms are described in [7, 8, 9, 12, 13, 19]. The second type is the non-contiguous area cartogram [15]. The regions have the true shape, but are scaled down and generally do not touch anymore. A third type of cartogram is based on circles and was introduced by Dorling [6]. Of particular relevance for this paper is the fourth type of cartogram, the *rectangular cartogram*, introduced by Raisz in 1934 [16], where each region is represented by a rectangle. This has the advantage that the areas (and thereby the associated values) of the regions can be easily estimated by visual inspection. Algorithms for such cartograms are described in [10, 18, 21].

Whether a cartogram is good is determined by several factors. One of these is the *cartographic error* [7], which is defined for each region as $|A_c - A_s| / A_s$, where A_c is the area of the region in the cartogram and A_s is the specified area of that region, given by the geographic variable to be shown. Other important factors are correct adjacencies and shapes of the regions, and suitable relative positions.

A purely rectangular cartogram can not always have both zero cartographic error and correct adjacencies. Consider the example in Figure 1. On the left you see the input map with specified area requirements (in brackets); the four grey rectangles on the outside should keep their sizes.

Rectilinear cartograms are a generalization of rectangular cartograms where regions can be rectangles, or L-shapes, or any other type of rectilinear polygon. Recently we proved [4] that in theory it is always possible to construct rectilinear cartograms with zero cartographic error and correct adjacencies. Figure 1 (right) shows a rectilinear cartogram with

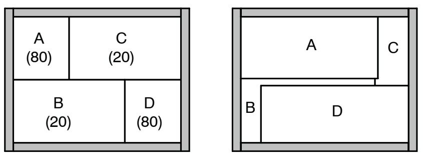

Figure 1: An input for which no rectangular cartogram has zero error and correct adjacencies, and a rectilinear cartogram for this input.

correct adjacencies and zero cartographic error for the input depicted in Figure 1 (left).

The (constructive) proof presented in [4] guarantees the existence of a rectilinear cartogram for any input map and any set of area values. However, the resulting regions can be quite complex, with thin "tails" that facilitate correct adjacencies. Here we develop a more practical variant of the approach proposed in [4]. Our new algorithm follows the general strategy of our previous method, but we introduce substantial algorithmic modifications in every step. We implemented and tested our algorithm on various data sets. It produces regions of very small complexity—in fact, most regions are rectangles—while still ensuring both exact areas and correct adjacencies for all regions.

One of the steps for which we develop a new technique is the computation of a *binary space partition* for a rectilinear map, which we now discuss in more detail.

Binary Space Partitions. Suppose we have a collection S of objects in the plane. A *binary space partition*, or *BSP* for short, for S is a recursive subdivision of the plane by splitting lines, until each cell of the final subdivision is intersected by a single, or perhaps a small number, of objects from S. BSPs are well known indexing structures [20, 17] that can be used to do point location, to answer range queries, and so on. When the objects in S are the regions of a map, as it will be the case in our application, we thus require that each cell be contained in a unique region of the map—see Figure 2 for an example. When we are dealing with BSPs for a rectilinear map, it is natural to also make the BSP rectilinear, that is, to require that the splitting lines be either horizontal or vertical. From now on, we limit our discussion to rectilinear BSPs for rectilinear maps.

The splitting lines of a BSP may cut the map regions into fragments. When this happens often, the performance deteriorates: the size of the BSP—and, hence, the storage needed to store it—increases, and algorithms working on the BSP slow down. Hence it is desirable to limit the fragmentation as much as possible and indeed several papers present algorithms to construct BSPs of small size. For example, D'Amore and Franciosa [3] show that any map consisting of n rectangles admits a BSP whose size (measured as the number of cells in the final subdivision) is at most $4n$.

However, experimental evidence shows that many rectangular maps admit BSPs of size close to n. Thus the question arises: it is possible, given a rectilinear map, to construct a BSP whose size is optimal for that particular map? We answer this question in the affirmative: we give a polynomial-time algorithm to construct a BSP of minimum size for any given rectilinear map. Our algorithm is quite general—it can compute optimal BSPs for a wide class of cost functions. When computing cartograms we make use of this generality. We apply a dedicated cost function leading to BSPs amenable to the constructing of high-quality cartograms.

Organization. In Section 2 we present our algorithm for computing optimal BSPs for rectilinear maps. In Section 3 we outline the approach from [4] and describe the modifications and new techniques introduced by our algorithm. We report on experimental results in Section 4.

2 Optimal BSPs for rectilinear maps

A *map* is a partition of a rectangle into a finite set of interior-disjoint regions. A *rectilinear map* is a map where every region is a rectilinear polygon. Let $\mathcal{M}$ be a rectilinear map with n edges in total. A BSP for $\mathcal{M}$ can be modeled as a BSP tree $\mathcal{T}$. Each internal node of $\mathcal{T}$ stores a splitting line and each leaf corresponds to a cell in the final BSP subdivision (see for example Figure 2).

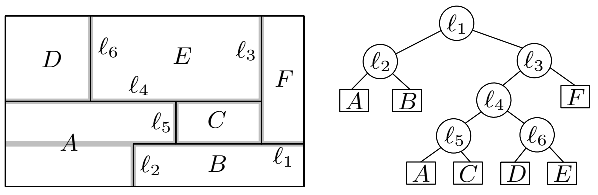

Figure 2: A BSP and the corresponding BSP tree. The leaves in the tree are labeled with the name of the region that contains the corresponding cell.

Our algorithm to compute optimal BSPs can handle different optimality criteria. For example, it can be used to compute a minimum-size or a minimum-depth BSP. Before we present the algorithm, we first describe the type of cost functions that our algorithm can handle.

Let $\mathcal{T}$ be a BSP tree for $\mathcal{M}$. We define the cost of a node in $\mathcal{T}$ as follows.

- We assume that each region r of $\mathcal{M}$ has a non-negative cost associated to it, denoted $\text{cost}(r)$. The costs of the regions in the map determine the costs of the leaf nodes of $\mathcal{T}$: for a leaf μ we define $\text{cost}(\mu) := \text{cost}(r_\mu)$, where r_μ is the unique region of $\mathcal{M}$ that contains the cell in the BSP subdivision corresponding to μ.
- The cost of an internal node ν is determined by the costs of its children and a function $F : \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$: if ν_1 and ν_2 denote the children of ν then we have $\text{cost}(\nu) := F(\text{cost}(\nu_1), \text{cost}(\nu_2))$.

The cost of a tree $\mathcal{T}$ is simply defined as the cost of its root. We call a BSP tree $\mathcal{T}$ for $\mathcal{M}$ *optimal* if its cost is minimal over all possible BSPs for $\mathcal{M}$. The goal of our algorithm is to compute such an optimal BSP tree, given the map $\mathcal{M}$, the cost function on the regions of $\mathcal{M}$ and a function F. In order for our algorithm to work, the function F needs to be monotone in the following sense:

Monotonicity: For any a, a', b, b' with $a \leq a'$ and $b \leq b'$ we have $F(a, b) \leq F(a', b)$ and $F(a, b) \leq F(a, b')$.

There are many natural optimality criteria that can be modelled like this. Suppose, for instance, that we want to compute a BSP of minimum size. Then we set the cost of each region to 1 and we define $F(a, b) = a + b$. To obtain a BSP of minimum depth we can also set the cost of each region to 1 but take $F(a, b) = \max(a, b) + 1$. Note that in both cases F is monotone. The possibilities of assigning different costs to different regions allows us to favor certain regions to be cut over other regions; in the application to cartograms we will make use of this flexibility.

We are now ready to describe our algorithm for computing an optimal BSP. Let $x_1, x_2, \ldots, x_{n_x}$ be the sorted sequence of distinct x-coordinates of vertical edges in $\mathcal{M}$, and let $y_1, y_2, \ldots, y_{n_y}$ be the sorted sequence of distinct y-coordinates of horizontal edges in $\mathcal{M}$. We first *normalize* the map: we replace the coordinates $x_1, x_2, \ldots, x_{n_x}$ by their ranks $1, \ldots, n_x$ and we replace $y_1, y_2, \ldots, y_{n_y}$ by $1, \ldots, n_y$. Note that an optimal (rectilinear) BSP for the original map corresponds to an optimal BSP for the normalized map; this is true because the cost of a leaf depends only on the cost of the map region that contains the leaf cell.

From now on, we use $\mathcal{M}$ to denote the normalized map. Observe that there exists an optimal BSP for $\mathcal{M}$ such that each splitting line contains (a part of) an edge of $\mathcal{M}$. Indeed, if there is a splitting line that does not contain a part of an edge it can be shifted until it does; it is not difficult to prove that this cannot increase the cost of the BSP.

Now consider an optimal BSP tree $\mathcal{T}^*$ all of whose splitting lines contain a part of some edge. The splitting line at the root cuts $\mathcal{M}$ into two "submaps". These submaps are again cut into smaller submaps, and so on. Because all splitting lines contain a part of an edge of $\mathcal{M}$, the submaps that arise during the process are always rectangles of the form $[x_1 : x_2] \times [y_1 : y_2]$, where x_1 and x_2 are x-coordinates of vertical edges in the map, and y_1 and y_2 are y-coordinates of horizontal edges. This leads us to define for $1 \leq x_1 < x_2 \leq n_x$ and $1 \leq y_1 < y_2 \leq n_y$ the following quantity:

$\mathrm{Opt}(x_1, x_2, y_1, y_2) :=$ the minimum cost of a BSP tree for the submap of $\mathcal{M}$ inside the rectangle $[x_1 : x_2] \times [y_1 : y_2]$.

Now consider an optimal BSP $\mathcal{T}^*$ that cuts a map into two smaller submaps. Because of the monotonicity of F, the two subtrees of the root of $\mathcal{T}^*$ must be optimal BSP trees for these two submaps. Thus we can get an optimal BSP by trying the different ways to cut $\mathcal{M}$ into submaps along an edge, and then for each such cut compute the optimal BSP tree for the two submaps.

Lemma 1
If $x_2 = x_1 + 1$ and $y_2 = y_1 + 1$ then $\mathrm{Opt}(x_1, x_2, y_1, y_2) = \mathrm{cost}(r)$, where r is the region of $\mathcal{M}$ containing the rectangle $[x_1 : x_2] \times [y_1 : y_2]$. Otherwise, we have

$$\mathrm{Opt}(x_1, x_2, y_1, y_2) =$$

$$\min(\min_{x_1 < x < x_2} F(\mathrm{Opt}(x_1, x, y_1, y_2), \mathrm{Opt}(x, x_2, y_1, y_2)) \\ \min_{y_1 < y < y_2} F(\mathrm{Opt}(x_1, x_2, y_1, y), \mathrm{Opt}(x_1, x_2, y, y_2)))$$

Using Lemma 1 we can compute an optimal BSP for a map $\mathcal{M}$ by dynamic programming. We have a 4-dimensional table $\mathrm{Opt}[1..n_x - 1,\ 2..n_x,\ 1..n_y - 1,\ 2..n_y]$ that we fill in as follows: we first determine $\mathrm{Opt}[x, x+1, y, y+1]$ for each $1 \leq x < n_x$ and $1 \leq y < n_y$ by finding for each rectangle $[x : x+1] \times [y, y+1]$ the map region containing it, and then we fill in the rest of the table in a bottom-up manner, using Lemma 1. We spend $O(n)$ time to compute the value for each entry in the table. At the end the optimal cost of a BSP tree for $\mathcal{M}$ is stored in $\mathrm{Opt}(1, n_x, 1, n_y)$. Once the table is filled in, we can make a second (top-down) pass through the table to produce a BSP tree that leads to the optimal cost; how to do this is standard in dynamic-programming algorithms, and so we omit the details. We obtain the following result.

Theorem 2 *Let $\mathcal{M}$ be a rectilinear map with n edges in total, and suppose we have a cost function on BSP trees determined by costs on the regions in $\mathcal{M}$ and a monotone function F, as described above. Then we can compute an optimal BSP for $\mathcal{M}$ in $O(n^5)$ time.*

Improvements. Our algorithm to compute optimal BSPs works fine for small maps—that is, maps with not too many regions—but for large maps it becomes quite slow and needs a lot of storage. Next we describe some heuristics that improve its performance.

Recall that there is an optimal BSP where each splitting line contains (a part of) an edge of $\mathcal{M}$. Hence, when we compute $\mathrm{Opt}(x_1, x_2, y_1, y_2)$ according to Lemma 1, we need to try only those values of x for which there is a vertical edge of the map $\mathcal{M}$ that lies at least partially within the rectangle $[x_1 : x_2] \times [y_1 : y_2]$. We can reduce the number of different y-values to be tested similarly.

When $F(a, b) = a + b$ then we can speed things up even further by using the following observation. Suppose we have a submap for which we want to compute an optimal BSP, and suppose that there is a region that has an edge cutting completely through the submap. Then a splitting line containing that edge will not cut *any* region of the submap. We call this a *free split*. In Figure 2, for instance, after we have split along ℓ_1, there is a free split in the submap above ℓ_1, namely along ℓ_3. When $F(a, b) = a + b$, we can always take such a free split without loosing optimality; we do not have to try any other options for the splitting line. (This is not the case in general. If we want to optimize the depth, for example, a free split can be unbalanced and not lead to an optimal result.)

If we apply these two heuristics, then in many cases we do not have to test all x with $x_1 < x < x_2$ and all y with $y_1 < y < y_2$. This reduces the computation time significantly. We can also use this to reduce the amount of storage. If there are cases where we do not try all possibilities according to Lemma 1, then the corresponding entry in the table $\mathrm{Opt}[1..n_x - 1,\ 2..n_x,\ 1..n_y - 1,\ 2..n_y]$ is apparently not needed. Indeed, in our experiments we observed that only 21% or less of the table was actually used.

Hence, we can reduce the storage substantially if instead of using the table $\mathrm{Opt}[1..n_x - 1,\ 2..n_x,\ 1..n_y - 1,\ 2..n_y]$, we store the values of the entries in a hash table. More precisely, we have a hash table $T[0..m]$ and a hash function h that maps a four-tuple (x_1, x_2, y_1, y_2) with $1 \leq x_1 \leq n_x - 1$, $2 \leq x_2 \leq n_x$, $1 \leq y_1 \leq n_y - 1$, and $2 \leq y_2 \leq n_y$ to an index in the range $0..m$. The value for $\mathrm{Opt}(x_1, x_2, y_1, y_2)$ will then be stored in $T[h(x_1, x_2, y_1, y_2)]$. To make this work, we need to use a top-down version of the dynamic-programming algorithm that uses memoization [2]. This also allows us to perform *pruning* to avoid solving subproblems of which we know that they cannot lead to optimal results.

Due to space restrictions we omit further implementation details and experimental comparisons with other BSP construction algorithms from this version of the paper. Some results concerning the performance of the optimal BSP algorithm can be found in Section 4.

3 Computing Rectilinear Cartograms

Recall that a map is a partition of a rectangle into a finite set of interior-disjoint regions. The *dual graph* $\mathcal{G}(\mathcal{M})$ of a map $\mathcal{M}$ is the graph that has one node per region and connects two regions if they are adjacent. Thus a cartogram has correct adjacencies if and only if its dual graph is the same as the dual graph of the original map. Given a graph $\mathcal{G}$, a *rectangular dual* of $\mathcal{G}$ is a rectangular map $\mathcal{M}$ (a map whose regions are rectangles) such that $\mathcal{G}(\mathcal{M}) = \mathcal{M}$.

Algorithmic outline. As mentioned before our algorithm follows the general outline of the algorithm we presented earlier [4], which we will refer to as the GD05-algorithm.

Our algorithm roughly works as follows: we construct a suitably modified version of the dual graph of $\mathcal{M}$, assign to each vertex in this graph $\mathcal{G}$ a weight that equals the required area of the corresponding region, and run the GD05-algorithm on the resulting vertex-weighted graph. This will lead to a correct rectilinear cartogram with regions of bounded complexity. To obtain good results in practice, however, we need to modify the GD05-algorithm in several places. We now first describe the various steps of our algorithm and then we discuss each of these steps in more detail.

```
Algorithm CARTOGRAM CONSTRUCTION($\mathcal{M}$)
Input. A map $\mathcal{M}$ and the required area for each region.
Output. A rectilinear cartogram $\mathcal{C}$.
1. Augment $\mathcal{M}$ with sea and pole vertices and compute
   the dual graph $\mathcal{G}(\mathcal{M})$. If necessary, modify the graph
   so that it has a rectangular dual.
2. Construct a rectangular dual $\mathcal{M}_1$ of $\mathcal{G}$.
3. Construct a BSP tree $\mathcal{T}$ of $\mathcal{M}_1$.
4. Move the BSP lines to correct the areas.
5. Grow tails to correct any broken adjacencies.
6. Move the BSP lines to repair the area errors re-
   introduced in Step 5.
7. Produce the final cartogram $\mathcal{C}$.
```

Step 1: Connectivity graph construction

The GD05-algorithm. This step was not needed in the GD05-algorithm, because it assumes that a vertex-weighted graph is given as input.

Our algorithm. To convert the input map into a suitable weighted graph, we follow the same approach as Van Kreveld and Speckmann [21]. We first compute the dual graph $\mathcal{G} = \mathcal{G}(\mathcal{M})$ of the input map and then assign to each vertex a weight that is the required area of the corresponding region in the map. For the subsequent steps of the algorithm it is convenient to work with a graph that has a rectangular dual. A planar graph has a rectangular dual with four rectangles on the boundary if and only if the following three conditions are met [1, 14]: (i) every interior face is a triangle, (ii) the exterior face is a quadrangle, and (iii) the graph has no separating triangles, that is, no 3-cycles for which there are vertices outside and inside the cycle.

To enforce condition (i), we simply triangulate any interior face that is not a triangle. (In most cases $\mathcal{G}$ is in fact already triangulated except for its outer face and for inner seas that are adjacent to several regions.) To enforce (ii), we add NORTH, EAST, SOUTH, and WEST vertices—we call these vertices *poles*—to form the boundary of the graph. There are several possibilities to enforce (iii). Here we add additional neighbors to all vertices of degree three or less. For example, when constructing the connectivity graph for the map of Europe, we split Belgium into two parts and make Luxemburg adjacent to both of them.

Furthermore, to make the future cartogram resemble the original map more closely, we add so-called sea vertices to our graph, representing the main bodies of water of the map. Some bodies of water are represented by more than one vertex to make the cartogram more recognizable or to increase the degree of a vertex where needed. After Step 1 we have a vertex-weighted graph $\mathcal{G}$ that admits a rectangular dual.

Step 2: Constructing a rectangular dual

The GD05-algorithm. The GD05-algorithm uses an algorithm by Kant and He [11] to construct the rectangular dual $\mathcal{M}_1$ for $\mathcal{G}$.

Our algorithm. We also use the algorithm by Kant and He, but—as proposed in [21]—we make use of the fact that it has the possibility to specify for any two neighboring vertices u and v whether the region corresponding to u should be north, east, south, or west of the region corresponding to v. This is done by labeling each edge with a direction: north, east, south, or west. In the GD05-algorithm it was not relevant how this was done since the input was a graph and not a map, but we want the edge labeling to correspond to the geographic situation as much as possible. For example, the label of the edge connecting the vertex for the Netherlands to the vertex for Germany should indicate that the Netherlands lies west of Germany. To determine how to label an edge, we employ a simple heuristic: we consider the relative positions of the centers of mass of the two regions in the original map $\mathcal{M}$. Often the relative positions clearly indicate how to label an edge, but sometimes two options are possible. For example, the center of mass of Germany lies north-west of the center of mass of Austria. This gives us a so-called *layout option*. If the input map $\mathcal{M}$ has m layout options, then there are 2^m different directed edge labeling possible, each leading to a different rectangular dual. Our algorithm tries all possible layout options and generates a rectangular dual for each realizable option. For every rectangular dual we then execute the remaining steps of the algorithm, leading to a number of different cartograms. From these, we can choose the best based on various quality criteria.

Step 3: Constructing a BSP

Step 3 computes a rectilinear BSP for the rectangular map $\mathcal{M}_1$ that results from Step 2. In general, we would like the BSP to avoid cutting regions as much as possible, because regions that are cut by the BSP have more complicated shapes in the final cartogram.

The GD05-algorithm. The GD05-algorithm uses the BSP-construction algorithm by d'Amore and Franciosa [3], which guarantees that each rectangle in $\mathcal{M}_1$ is cut into at most four pieces.

Our algorithm. We use the optimal BSP construction algorithm described in Section 2 and we experimented with several different optimality criteria. Recall that the cost of a BSP tree is determined by the costs assigned to each of the regions—this determines the costs of each leaf in the BSP tree—and a function F that determines how to compute the cost of a node from the costs of its children. In our experiments we always used $F(a, b) = a + b$. This means that the cost of a BSP is simply the sum over all regions of the number of pieces into which the region is cut, weighted by the cost of that region. For the cost of cutting a region, we tried several alternatives:

- The obvious choice is to set the cost of each land region to 1. This minimizes the total number of cuts over all regions, leading to cartograms with regions that have simple shapes. Because the final shape of the sea regions is less important, we give them cost zero.

Observe that a region that is cut by the BSP has more flexibility to change its shape in the remaining steps. This suggests to give "difficult" regions a lower cost, which ensures they are cut first if any regions need to be cut at all. This can be done in two ways:

- Regions whose required area deviates greatly from their current area are more difficult to handle, so we can give such regions a lower cost. More precisely, if A_r is the required area of a land region and $A_{\mathcal{M}_1}$ is the area of that region in $\mathcal{M}_1$ then we set its cost to be
$$\min\left(\frac{A_r}{A_{\mathcal{M}_1}}, \frac{A_{\mathcal{M}_1}}{A_r}\right) .$$
- Regions with many neighbors might need some extra flexibility. Thus we set the cost of a land region to
$$\frac{1}{1 + \text{number of land neighbors}} .$$

We compared these BSP construction strategies experimentally to the following other strategies.

- The algorithm by D'Amore and Franciosa [3], as used in [4]. We also tried a variant of this algorithm that applied free splits (see the previous section) whenever possible.
- A simple greedy strategy that always uses a splitting line for which the total cost of the cut regions is minimal. The greedy strategy automatically uses free splits if they are possible, because a free split has zero cost. As in our optimal BSP algorithms, we define the cost of cutting a sea region to be zero and we try different costs for the land regions: cost 1 for each region, a cost that depends on the deviation of the original and specified area of the region, and a cost that depends on the number of neighbors of the region. When there are more splitting lines with minimum cost, we take the one that is most balanced (in terms of the number of regions on both sides of the line).

Every BSP construction method might cut some regions into two or more subregions and then the weight w of the region has to be distributed between its k subregions. In other words, we must assign positive weights $w_1, \ldots, w_k$ to the subregions such that $\sum_{i=1}^{k} w_i = w$. We do this proportionally to the areas as it is also done in the GD05-algorithm: if a rectangle r from $\mathcal{M}_1$ is cut by the BSP into subrectangles $r_1, \ldots, r_k$, then the specified area for a subrectangle r_i is (required area of r) $\cdot$ (area(r_i)/ area(r)).

Step 4: Getting the areas right

The input to Step 4 is a BSP tree $\mathcal{T}$ for the rectangular dual $\mathcal{M}_1$. Recall that each internal node ν in a BSP tree stores a splitting line; we denote this line by $\ell(\nu)$. Also recall that each leaf in the BSP corresponds to a cell in the BSP subdivision (which in our case is contained in one of the regions of $\mathcal{M}_1$). For an internal node ν we define $R(\nu)$ to be the union of all the cells corresponding to leaves in the subtree of ν. This implies that $R(\text{root}(\mathcal{T}))$ is simply the whole map area, and that the splitting line $\ell(\nu)$ splits $R(\nu)$ into $R(\text{leftchild}(\nu))$ and $R(\text{rightchild}(\nu))$.

The GD05-algorithm. The GD05-algorithm traverses $\mathcal{T}$ top down. At each node ν the splitting line $\ell(\nu)$ is repositioned while maintaining the following invariant: when a node ν is handled, the current area of $R(\nu)$ is equal to the sum of the weights of the required areas of the leaf cells in the subtree $\mathcal{T}(\nu)$ rooted at ν. We start at the root and scale $\mathcal{M}_2$ for R to have the required area. Then for each internal node ν we position $\ell(\nu)$ such that $R_{\text{rightchild}(\nu)}$ and $R_{\text{leftchild}(\nu)}$ have correct areas. When we arrive at the leaves the corresponding cells have correct areas.

The repositioning of the splitting lines may destroy some of the adjacencies. This will be remedied in the later steps.

Our algorithm. Because the repositioning of splitting lines can destroy adjacencies, we would like to move each splitting line as little as possible. But unfortunately, the location of each splitting line is completely determined by the required areas. We observe, however, that this is not completely true: we can make use of the fact that sea regions are introduced artificially and do not have specified area requirements. Initially, we set the required areas for the sea rectangles such that the total sea area will sum up to some fixed percentage (for example, 20%) of the total map area, and then we distribute this total sea area over the various sea rectangles in a more or less arbitrary manner. Now, instead of preserving these initial settings, we do the following. Suppose that we are repositioning a vertical splitting line $\ell(\nu)$ and that we have to move it to the right. If there are sea rectangles to the left and to the right of $\ell(\nu)$, then we can decrease the required area for the sea to the left of $\ell(\nu)$ and increase the required area for the sea to the right of $\ell(\nu)$. The total sea area stays the same, but $\ell(\nu)$ has to be moved less, or maybe not at all. Hence, we reduce the movement of the BSP lines and lower the number of broken adjacencies. However, if we transfer too much sea area, then the final map will not look good. Therefore we use a threshold parameter that gives an upper bound on the percentage of the sea area that we are allowed to transfer.

Step 5: Tailing

Repositioning the splitting lines during the previous step may have caused some of the adjacencies to be broken, as illustrated in Figure 3. Step 5 fixes these adjacencies.

The GD05-algorithm. The GD05-algorithm fixes the adjacencies using so-called *tails*. These are very thin rectangles that are added to a region to connect it to some other region. The region then becomes an L-shape—see Figure 3—or something more complicated if a region gets several tails.

Every tail introduces area errors. Therefore the tails created by the GD05-algorithm are extremely thin, to guaran-

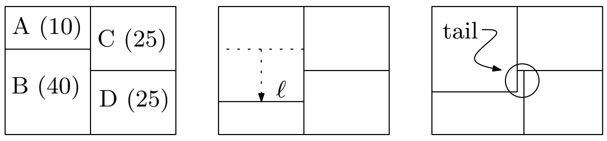

Figure 3: A rectangular dual $\mathcal{M}_1$ with specified areas. The splitting line ℓ is moved down in Step 4 and the adjacency between B and C is broken; it is restored by adding a tail to B.

tee that the errors can be repaired in Step 6 without destroying any adjacencies.

Our algorithm. We follow the same approach as the GD05-algorithm, but we add two heuristics: one is aimed at reducing the number of tails, the other at increasing the width of the tails. We reduce the number of tails by ignoring some of the broken adjacencies without compromising the correctness of the cartogram. This is possible when some of the sea areas of the map are represented by more than one sea region of the cartogram, because it is not important which one of these sea regions a land region is bordering. Thus we can relax the adjacency requirements as follows:

- The sea regions forming the same sea area on the map have to form a connected region, but which sea rectangles border each other is not important.
- Each land region must have the correct adjacencies with the other land regions, but when it is adjacent to a sea region then it does not matter which sea region (of the same sea area) it is adjacent to.

Our second heuristic tries to increase the width of the tails. This is important because the tails that are produced by the GD05-algorithm are so extremely thin that they easily become invisible. What follows is a rough sketch of our approach. We start with fairly wide tails and run the algorithm. If it finishes with a correct cartogram then we are done. But the tails might be too wide to be able to repair the areas in Step 6 without breaking adjacencies. When this happens then we decrease the tail width and try again. In fact, our implementation is a bit more careful and reduces the tail width only locally (namely, at the place in the map where the algorithm could not repair the area).

Step 6: Correcting the areas again

The GD05-algorithm. The errors in the areas are corrected by moving the splitting lines —which have by now become polylines—of the BSP, similar to Step 4. This time, however, the errors to be repaired are so small that one can prove that the splitting lines have to be moved only so little that no adjacencies are destroyed.

Our algorithm. We follow the same approach as the GD05-algorithm, but we use the extra flexibility that the sea areas give us, as described in Step 4.

Step 7: Producing the final cartogram.

Some of the regions may have been split into parts when constructing the connectivity graph or the BSP. We merge these pieces so that they form one region again.

4 Implementation and test results

Recall that rectangular cartograms can not always achieve both zero cartographic error and correct adjacencies, while our rectilinear cartograms always achieve both. Hence our main quality criterion is the shape of the regions. We want to use as many rectangular regions as possible, and only a few L-shapes or other regions of higher complexity. Furthermore, we would like to avoid very thin regions. We measure the complexity of a region by the number of corners and we generalize aspect ratio to rectilinear polygons by defining the fatness of a rectilinear polygon R as

$$\text{area}(R)/\,\text{area}(\text{bounding square of } R).$$

Note that fatness is the inverse of aspect ratio when R is a rectangle. We also measure the width of the tails that are added in Step 5 of our algorithm: the GD05-algorithm uses extremely thin tails—so thin that they are invisible when printed—and we want to see whether our modification leads to better results. Finally, since BSPs are useful in other contexts as well, we also report on the size of the BSP produced in Step 3 of our algorithm.

We use two different maps: the countries of Europe and the states of the US. Our data sets are area, population, total highway length, gross domestic product (GDP; Europe only), native population (US only), number of cars (US only), and number of farms (US only). Thus in total we performed 10 different experiments. As mentioned earlier our algorithm tries a number of different layout options in Step 2. The results we report below are always for the layout with the largest minimum tail width. For each experiment we report the following measures:

- maximum complexity (MC) and average complexity (AC) of the regions in the final cartogram.
- minimum fatness (MF) and average fatness (AF) of the regions in the final cartogram.
- minimum tail width (TW) of the tails added in Step 5.
- maximum number (MP) and average number (AP) of rectangular pieces per map region after Step 3.

We tried several variants of our algorithm, which only differ in the way the BSP is constructed in Step 3. They are:

- The optimal algorithm from Section 2, with three different cost functions for the regions: unweighted (CN), weighted based on area deviation (AD), and weighted based on the degree (Deg).
- A greedy algorithm that always uses a splitting line for which the total cost of the cut regions is minimal, with the same three cost functions.
- The algorithm by D'Amore and Franciosa [3], as used by the GD05-algorithm. We use a variant of this algorithm that applies free splits whenever possible; otherwise the results are much worse.

Let us summarize the results of our experiments:

- In 8 out of 10 experiments, a variant (usually the one based on area deviation) of the optimal algorithm produced a cartogram with the lowest average complexity of the regions. In the two other experiments the greedy BSP algorithm was the best.

EU population							
QM	D'Am	Greedy			Optimal		
		CN	AD	Deg	CN	AD	Deg
MC	18	10	14	8	8	8	10
AC	5.65	5.1	5.05	4.9	4.6	4.6	4.6
MF	0.03	0.02	0.02	0.03	0.04	0.04	0.04
AF	0.45	0.45	0.47	0.47	0.45	0.44	0.45
TW	1.26	1.99	1.76	2.18	3.07	3.07	3.07
MP	5	3	3	3	3	3	3
AP	1.45	1.13	1.15	1.15	1.13	1.13	1.13

EU GDP							
QM	D'Am	Greedy			Optimal		
		CN	AD	Deg	CN	AD	Deg
MC	10	12	10	12	10	10	10
AC	5.2	4.95	4.65	4.85	4.6	4.55	4.55
MF	0.03	0.03	0.02	0.01	0.06	0.05	0.05
AF	0.40	0.37	0.42	0.40	0.41	0.44	0.44
TW	1.66	1.96	0.82	1.02	2.32	2.20	2.20
MP	3	3	3	3	3	3	3
AP	1.28	1.13	1.10	1.13	1.13	1.08	1.08

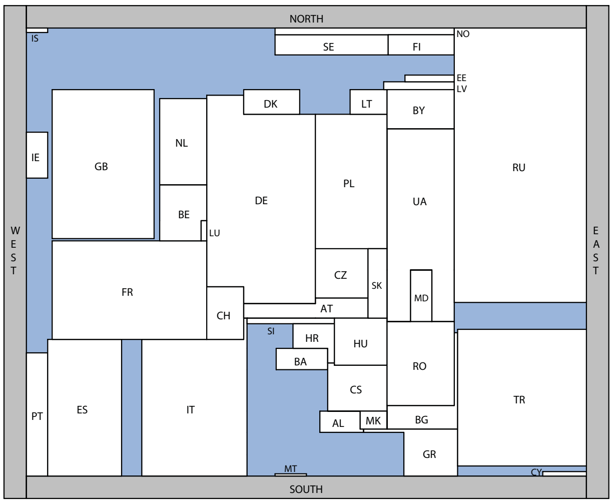

Figure 4: EU population.

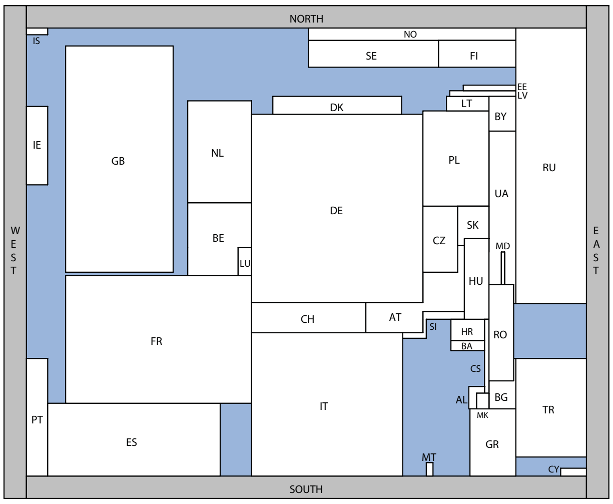

Figure 5: EU GDP.

- The average complexity of the regions in the best cartogram was never more than 5, and often below 4.6. Since a rectangle has complexity 4, this means that most of the regions are rectangles, as hoped.
- The maximum complexity of the regions in the best cartogram was never more than 10. This is significantly better than the maximum complexity of 20 guaranteed by the GD05-algorithm.
- The tails that are produced are clearly visible.
- The BSP algorithms cut only a few rectangles into pieces; most rectangles are never cut. More precisely, in all experiments there was a BSP such that no rectangle was cut into more than two pieces.
- Although each region of the cartogram might represented by several rectangles in the rectangular dual $\mathcal{M}_1$ (for example, Belgium is represented by 2 rectangles), the average number of rectangular pieces per map region after Step 3 was always 1.13 or less—much better than the average of 2 guaranteed by the algorithm of D'Amore and Franciosa [3].

Figures 4-6 show the exact results for some of our experiments. In all cases the total sea area was set to 20%. The cartogram in Figure 4 shows Europe with the theme population. Figure 5 is another cartogram of Europe, with the theme gross domestic product. Both Europe cartograms were produced with the unweighted optimal BSP algorithm. Figure 6 and Figure 7 show two cartograms of the US with the theme population and number of highways, respectively. Both were also produced by the optimal BSP algorithm, based on the number of neighbors.

Note that sometimes the unweighted optimal BSP seems to produce a higher number of pieces than the weighted optimal BSP, which would contradict the optimality of the unweighted algorithm. This is caused by the fact that we tried a number of layout options and report the data for the option with the widest tails. Thus the data reported for the weighted version can be based on a different layout than the data for the optimal algorithm. We did not try to optimize the running time of our algorithm, but it is reasonably fast nevertheless (less than a minute for a single layout option).

5 Conclusions

We presented an algorithm for constructing rectilinear cartograms with zero cartographic error and correct region adjacencies, and we tested our algorithm on various data sets. Our experimental results demonstrate that our cartograms have very small complexity, that is, most regions are rectangles and no region has more than 10 corners. Furthermore, the fatness of the regions is sufficient to accurately judge the area and all tails are wide enough to clearly indicate the region adjacencies.

We also introduced a polynomial-time algorithm for computing optimal BSPs for a given map. Our algorithm works for a general class of optimality criteria and our experimental results show that it generally constructs BSPs of significantly smaller size than previously known approaches.

Acknowledgements. We thank Xavier Clairbois for implementing the BSP algorithm.

US population							
QM	D'Am	Greedy			Optimal		
		CN	AD	Deg	CN	AD	Deg
MC	8	8	8	8	8	8	8
AC	5.04	5.29	5.17	4.79	5.17	5.17	5.25
MF	0.04	0.02	0.02	0.03	0.02	0.02	0.02
AF	0.41	0.37	0.38	0.40	0.37	0.37	0.39
TW	1.01	1.20	1.35	1.37	1.36	1.36	1.36
MP	2	2	2	2	2	2	1
AP	1.04	1.02	1.08	1.02	1.02	1.02	1.00

US highways							
QM	D'Am	Greedy			Optimal		
		CN	AD	Deg	CN	AD	Deg
MC	12	8	8	10	8	8	10
AC	5.29	4.58	4.67	4.79	4.71	4.67	4.67
MF	0.06	0.06	0.10	0.09	0.08	0.10	0.12
AF	0.47	0.50	0.49	0.50	0.48	0.45	0.47
TW	1.40	2.98	2.09	2.45	2.45	1.84	2.75
MP	2	1	2	2	2	2	2
AP	1.17	1.00	1.04	1.15	1.04	1.04	1.06

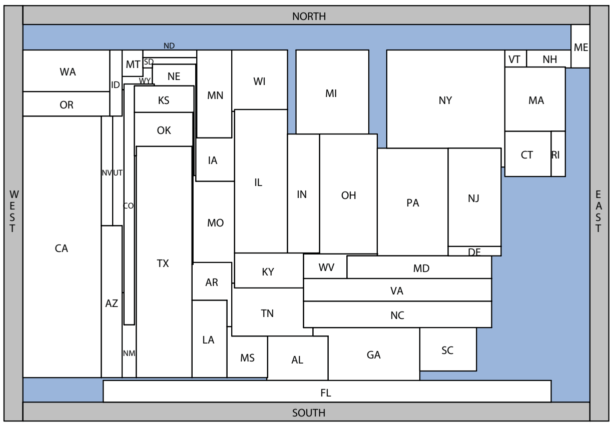

Figure 6: US population.

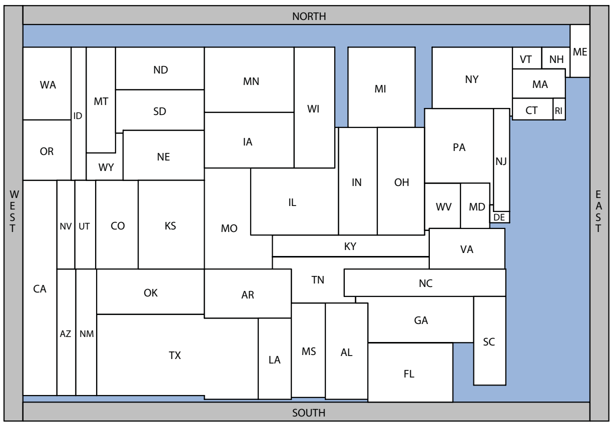

Figure 7: US highways.

References

[1] J. Bhasker and S. Sahni. A linear algorithm to check for the existence of a rectangular dual of a planar triangulated graph. *Networks*, 7:307–317, 1987.

[2] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2nd edition, 2001.

[3] F. d'Amore and P. G. Franciosa. On the optimal binary plane partition for sets of isothetic rectangles. *Information Processing Letters*, 44(5):255–259, 1992.

[4] M. de Berg, E. Mumford, and B. Speckmann. On rectilinear duals for vertex-weighted plane graphs. In *Proc. 13th Intern. Symposium on Graph Drawing*, number 3843 in LNCS, pages 61–72, 2005.

[5] B. Dent. *Cartography: Thematic Map Design.* McGraw-Hill, 5th edition, 1999.

[6] D. Dorling. *Area Cartograms: their Use and Creation.* Number 59 in Concepts and Techniques in Modern Geography. University of East Anglia, Environmental Publications, Norwich, 1996.

[7] J. A. Dougenik, N. R. Chrisman, and D. R. Niemeyer. An algorithm to construct continuous area cartograms. *The Professional Geographer*, 37(1):75–81, 1985.

[8] H. Edelsbrunner and E. Waupotitsch. A combinatorial approach to cartograms. *Computational Geometry: Theory and Applications*, 7:343–360, 1997.

[9] M. Gastner and M. Newman. Diffusion-based method for producing density-equalizing maps. *Proc. National Academy of Sciences of the United States of America*, 101(20):7499–7504, 2004.

[10] R. Heilmann, D. A. Keim, C. Panse, and M. Sips. Recmap: Rectangular map approximations. In *Proc. IEEE Symposium on Information Visualization*, pages 33–40, 2004.

[11] G. Kant and X. He. Regular edge labeling of 4-connected plane graphs and its applications in graph drawing problems. *Theoretical Computer Science*, 172:175–193, 1997.

[12] D. Keim, S. North, and C. Panse. Cartodraw: A fast algorithm for generating contiguous cartograms. *IEEE Transactions on Visualization and Computer Graphics*, 10:95–110, 2004.

[13] C. Kocmoud and D. House. A constraint-based approach to constructing continuous cartograms. In *Proc. 8th Intern. Symposium on Spatial Data Handling*, pages 236–246, 1998.

[14] K. Koźmiński and E. Kinnen. Rectangular dual of planar graphs. *Networks*, 5:145–157, 1985.

[15] J. Olson. Noncontiguous area cartograms. *Professional Geographer*, 28:371–380, 1976.

[16] E. Raisz. The rectangular statistical cartogram. *Geographical Review*, 24:292–296, 1934.

[17] H. Samet, editor. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1989.

[18] B. Speckmann, M. van Kreveld, and S. Florisson. A linear programming approach to rectangular cartograms. In *Proc. 12th Intern. Symposium on Spatial Data Handling*, pages 250–257, 2006.

[19] W. Tobler. Pseudo-cartograms. *The American Cartographer*, 13:43–50, 1986.

[20] M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors. *Algorithmic Foundations of Geographic Information Systems.* Springer, 1997.

[21] M. van Kreveld and B. Speckmann. On rectangular cartograms. In *Proc. 12th European Symposium on Algorithms*, number 3221 in LNCS, pages 724–735, 2004.