Mosaic cartograms for grid maps

Răducanu, Bogdan

Award date:
2021

Mosaic cartograms for grid maps

Bogdan Raducanu Student number: 0870280
b.raducanu@student.tue.nl

July 3, 2021

Abstract

Grid maps are used to display data coupled with a visual, spatial information. Depending on the amount of data, the spatial information has to be distorted for the sake of the data's visibility. In a grid map, each spatial element is condensed into a simple tile (such as a square or hexagon) and then arranged such that the global shape of all tiles matches the global shape of the input with the least amount of spatial distortion, such that each tile is visually identifiable according to its spatial information. We analyze the state-of-the-art solution for generating such grid maps and identify avenues of improvement which we pursue in this thesis. We improve two steps of this solution: the partitioning of an input map based on salient features and its conversion into a square tile mosaic cartogram. The main improvements over the state-of-the-art consist of guaranteeing a valid mosaic cartogram conversion. We discuss our methodology, provide an implementation which we apply to one of the datasets used by the state-of-the-art solution, a map of mainland Netherlands and its municipalities, compare it to the state-of-the-art result and discuss further avenues of improvement for our implementation.

Contents

Chapter 1

Introduction

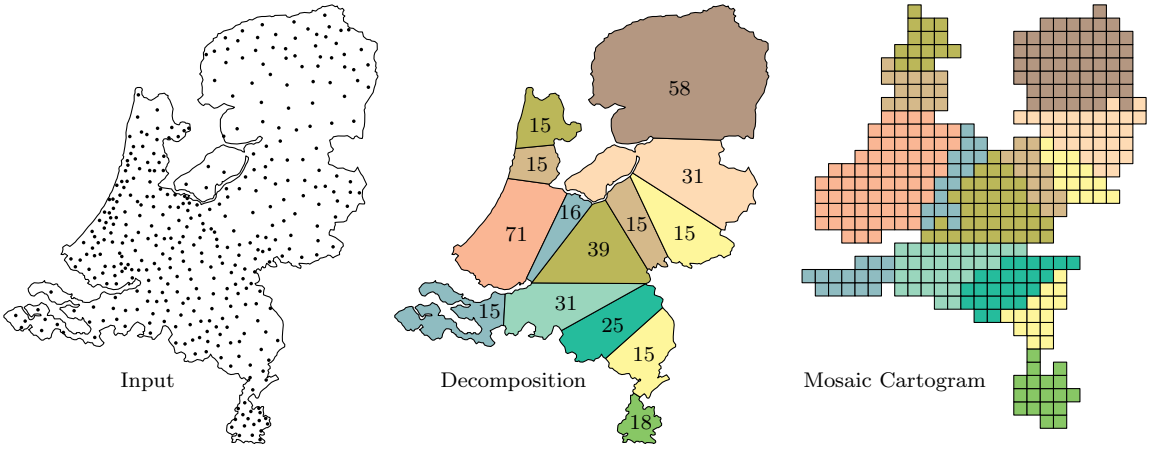


Figure 1.1: Our two main steps for computing a valid mosaic cartogram: decomposing the containing shape into parts that infer a binary tree dual graph, and computing a mosaic cartogram using these parts.

As technology evolves and data collection becomes more prevalent, the complexity and dimensionality of the data also increases with time. Thus, it is no surprise that understanding such data also requires increasing time and effort. However, not everyone has the time or is willing to invest the required effort to study such data in its raw form. And for a certain data complexity, comprehending it can become impossible for a human to achieve. It is clear that an extra step has to be taken before undergoing the study of such data, especially if it is wanted that the data should be quick and easily understood by anyone engaging with it. Concerned with this extra step is the field of data visualization, which pursues ways to aggregate, organize and visualize data of any given complexity.

One often encountered type of data that needs proper visualization is data that is coupled

with a spatial dimension. In this thesis, we focus on one particular way to show such data, namely, using maps. Maps are used every day for various purposes such as navigation, geolocation, entertainment and many others. A map consists of not only geographical features, but also certain data that are associated with these features, such as a label marking the name of a certain region, or a population count. Maps that focus more on the accurate location of the spatial data conveyed are called topographic maps. However, the more information we want to convey on a map, the more cluttered and hard to read it becomes. The field of cartography often has to tackle problems like this, in which it has to be decided which information to omit in order to keep the geographical representation of the map as faithful to the real world as possible, while also offering as much of the desired functionality of the map as possible. These types of maps where the focus is more on the properties of the spatial data are called thematic maps.

In certain situations, where no data that has to be conveyed on the map can be omitted, i.e. all of it has to be displayed, this becomes a problem. It is clear that the focus of data visualization is the readability of this data, thus, readability cannot be sacrificed. Thus, the only remaining compromise that can be done concerns the accuracy of the spatial information itself, the geographical features. In order to display large amounts of data on a thematic map, the real world representation of space would have to be distorted, in order to fit the desired data. However, the distortion should preserve as much real world information as possible in order for the spatial information, distorted as it may be, to still be recognizable to the reader, otherwise the entire functionality of a thematic map is lost.

One solution for maximizing readability while minimizing spatial distortion consists of the "grid map". A grid map is a thematic map in which each spatial element (singular point or a bounded region) is compacted into one specific geometrical shape of a certain size, and are arranged such that the global shape of this arrangement resembles the global shape of the spatial elements as much as possible. Each compacted spatial element is identical in shape and size and are referred to as *tiles*, and are laid out on a regular grid (such as a square tiling). In our case, the spatial elements do not consist of bounded regions, but rather singular points referred to as *sites*, as seen Figure 1.1's Input.

In this thesis, we revisit an existing solution for automatically generating grid maps. We identify an avenue of improvement within this existing solution, and we propose and implement a new solution for generating coherent grid maps, ultimately applying it to the dataset that the previous paper was focused on, a map of the Netherlands with its municipalities. One major improvement over the previous solution is that the wanted number of tiles and wanted regional adjacencies are guaranteed to be maintained in the grid map we generate.

We describe and illustrate each aspect of the our solution, and showcase the result of applying this solution on a real world dataset. We then further discuss its potential limitations as well as propose certain methods to address them.

1.1 Related work

A solution that tackles the grid maps problem directly is discussed and implemented by Meulemans et al. [1]. Taking a shape and a set of spatial elements contained within the given shape as input, their solution involves a 3-step pipeline for the automatic generation of grid maps (illustrated in Figure 1.2 from the source paper):

1. The decomposition of the given shape into smaller shapes, based on salient features (controlled by a variable called *dilation*), which infers a dual graph. The minimum number of sites that each small shape must contain is controlled by a variable called *productivity*.
2. The generation of a grid map precursor (referred to as a *mosaic drawing*) using as input the decomposed shapes and the number of spatial elements that each of the decomposed shape contains generated in the previous step. This step uses a solution proposed by Cano et al. [2].
3. The correspondence of each tile in the mosaic drawing with a spatial element via point-set matching, for the minimization of the squared Euclidean distance between a mosaic drawing tile and a spatial element, turning the mosaic drawing into a grid map.

While this solution does generate a grid map that preserves the global shape of the given input pretty well, the second step of the pipeline cannot guarantee that for all inputs it is able to preserve the desired number of tiles, or to guarantee that all adjacencies dictated by the decomposition step are correct. If the number of tiles aren't as specified, this means that at least one spatial element does not have a representation in the grid map (leading to loss of information) or there is at least one extra tile that does not map to any one spatial element (leading to introduction of noise). If the adjacencies given by the decomposition are not respected in the mosaic drawing, the resulting grid map would lose the proper representation of the decomposition's salient features. We propose

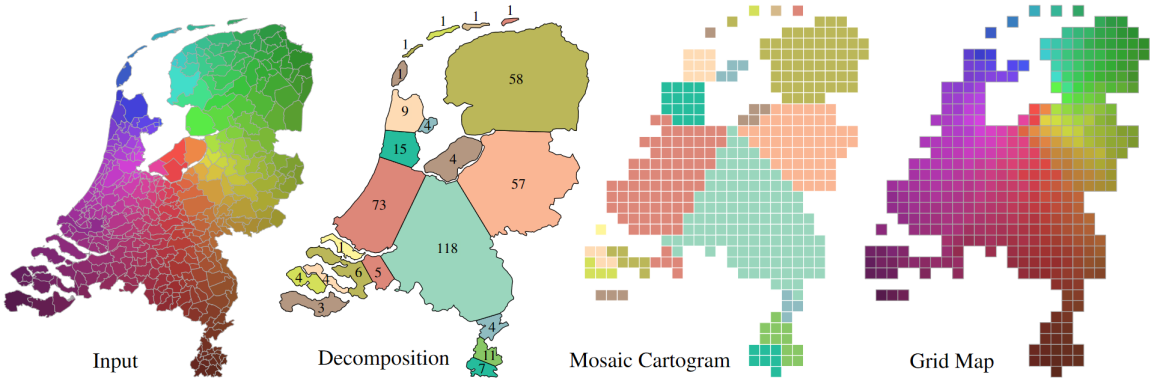


Figure 1.2: The three-step pipeline of Meulemans et al. [1], using the municipalities of the Netherlands as input. This figure is taken from the source paper.

a number of changes to this existing solution, such that a grid map does guarantee the previously mentioned criteria while also preserving the global shape of the input.

1.2 Reduction of pipeline complexity

The avenue of improvement comes from an observation made by the solution's authors themselves: they noticed that the decomposition step always produces a dual graph that is a tree. Considerable complexity can be reduced in the mosaic drawing generation step by making the assumption that the decomposition step produces tree graphs, since it does not have to handle all types of planar graphs. Additionally, by changing the decomposition step such that it creates a certain kind of tree graphs, it is possible to introduce the adjacency and tile number guarantee for all generated mosaic drawings. The final step of the pipeline remains unchanged and thus, it is not part of this thesis' scope.

1.3 Mosaic cartograms revisited

"Mosaic drawing" is a term coined by Cano et al. [2]. Mosaic drawings and mosaic cartograms have been well defined in their paper, and as such, certain definitions and notations that are relevant are borrowed and re-stated from that paper for consistency.

In order to understand mosaic drawings and mosaic cartograms, the concept of a tiling has to be introduced first. A tiling (or tesselation) of a plane is defined as the covering of a Euclidean plane with geometric shapes, called tiles such that there are no gaps in between the tiles and no tiles are overlapping eachother. Tilings can be periodic, in which using a certain type of geometrical shapes would create a repeating pattern, and aperiodic, in which the shapes involved are not able to create a repeating pattern. The relevant plane tilings to this problem are periodic tilings that use regular polygons of the same size as geometric shapes, such as triangles, squares, hexagons etc. which can be seen in Figure 1.3.

While the original solution by by Cano et al. [2] for generating mosaic drawings supports periodic regular tilings of square and hexagonal types, mosaic drawings in this thesis are only based on the square kind, referred to as a square tiling of a plane.

Let $\mathcal{T}$ be a square tiling of a plane. A configuration C is defined as a set of tiles of $\mathcal{T}$. A configuration is called contiguous if all tiles within it are edge-connected. A contiguous configuration is called simple if the tiles contained in the set are simply connected, i.e, the tiles do not contain a hole. Figure 1.4 shows these properties.

Two simple configurations C_1 and C_2 are called adjacent if there is at least one tile from C_1 that is edge-connected to at least one tile from C_2, exemplified in Figure 1.5. Additionally, a union of configurations is itself a configuration.

Let T be a tree where all n nodes are of at most degree 3, with k degree 3 nodes and a

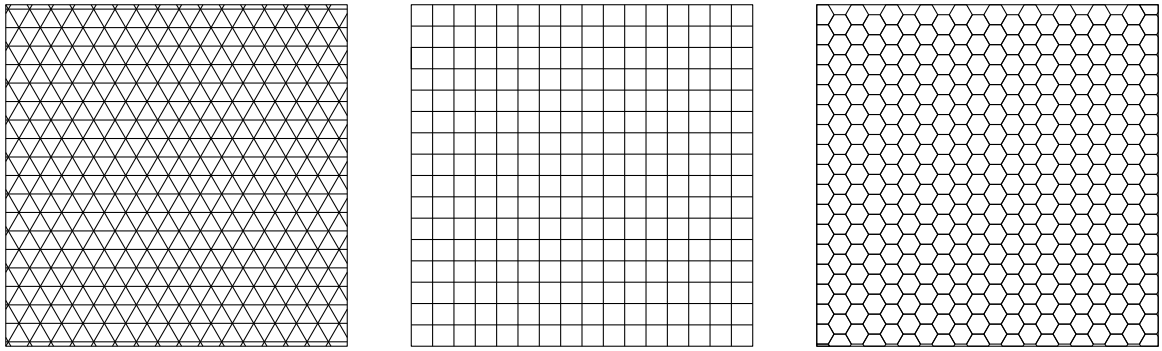


Figure 1.3: Examples of periodic regular tilings.

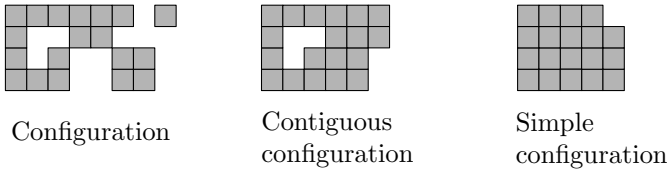


Figure 1.4: Types of configurations.

given embedding. Each node v in the given tree T contains an integer sq_v representing the number of tiles the configuration of v, denoted as $C(v)$, should contain. An arbitrary example of such T can be seen in Figure 1.6.

Let $S(V, h, t)$ be an induced path subgraph of T, where V is a subset of T's nodes and $h, t \in V$ being the head node and the tail respectively. We call this path graph S a *spine*

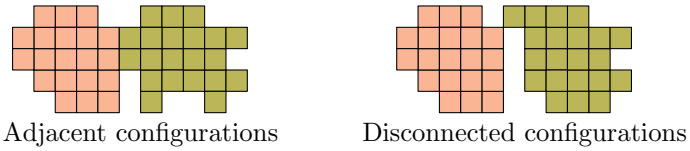


Figure 1.5: Adjacent versus non-adjacent configurations.

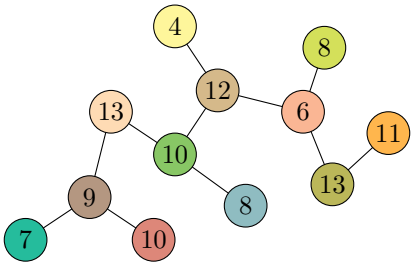


Figure 1.6: A tree T with the sq_v of each node.

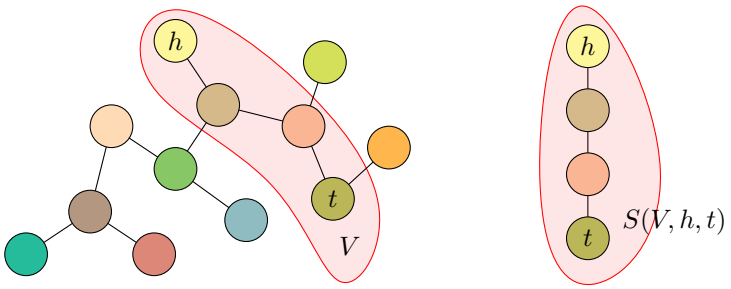


Figure 1.7: A spine $S(V, h, t)$ within T.

(As seen in Figure 1.7).

Let $B(V, r)$ be an induced subgraph of T rooted at node r, where V is a subset of T's nodes and r is a node of at most degree 2 within the induced subgraph. $B(V, r)$ is a rooted binary tree, and each node $v \in V$ is augmented with an integer k_v representing the number of degree 3 nodes contained by the subtree B_v rooted at v. Such a tree is exemplified in Figure 1.8.

We can now define a decomposition of T as follows: Let V_S be a subset of T's nodes with which a spine $S(V_S, h, t)$ can be made, where $h, t \in V_S$. For each node $u \notin V_S$ that is adjacent to a node $v \in V_S$, we create a subset V_u of T's nodes such that it contains all nodes whose path to u does not contain node v. For each such subset V_u, a binary tree $B(V_u, u)$ can then be made. In Figure 1.9, a decomposition of an arbitrary T is shown.

Thus, based on the decomposition of tree T, we can define T as a spine between two arbitrary nodes in T ($S(V_S, h, t)$, where h, t and the nodes between them are contained in V_S, and a set of rooted binary trees that would remain if the spine were to be remove ($B(V_u, u)$, where u is the root of the binary tree and V_u contains the nodes of the binary tree). Figure 1.10 illustrates an arbitrary tree T defined by a decomposition of itself.

We define a mosaic drawing $\mathcal{D}_{\mathcal{T}}(T)$ of tree T with n nodes as the set of all configurations

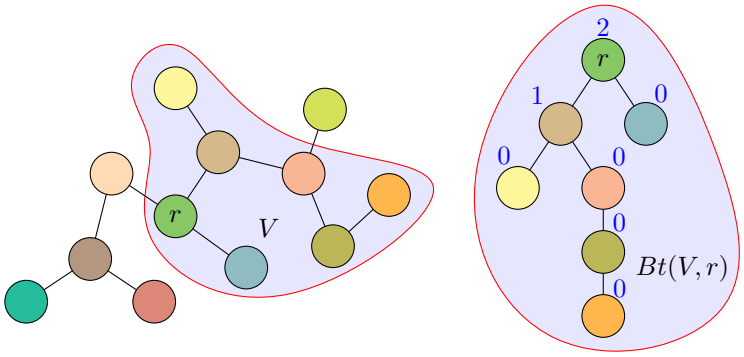


Figure 1.8: A binary tree $B(V, r)$ within T, with the k_v values of each node (in blue).

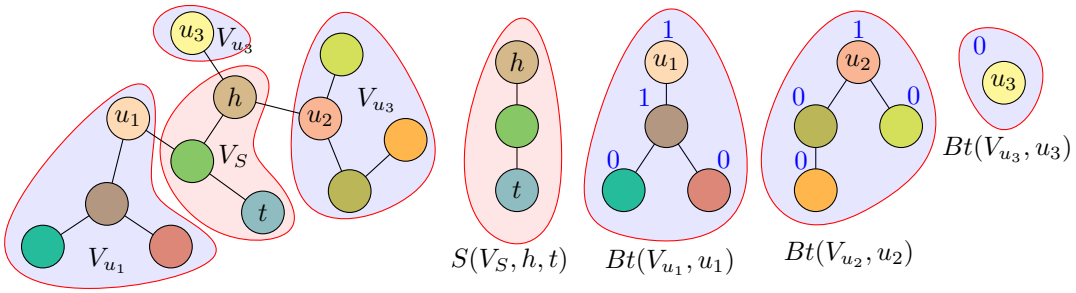


Figure 1.9: A decomposition of T into one spine and three binary trees, with the k_v values of each binary tree node (in blue).

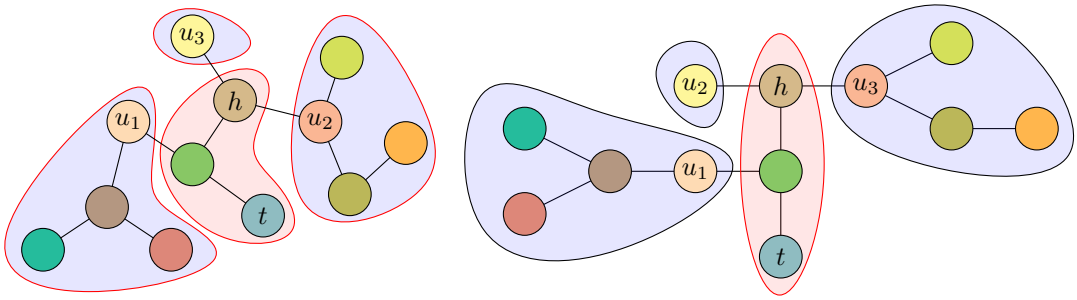


Figure 1.10: Redefining T based on a decomposition.

$C(v)$, where v is a node of T. We note that in a mosaic drawing, the number of tiles that $C(v)$ contains can be equal or different than sq_v.

We define a mosaic cartogram $\mathcal{C}_{\mathcal{T}}(T)$ of tree T with n nodes as the set of all configurations $C(v)$, where v is a node of T, as a mosaic drawing of T with the following properties:

- Each configuration $C(v)$ is a simple configuration.
- There are exactly sq_v tiles for each configuration $C(v)$.
- If two nodes v and u of T are adjacent, then their simple configurations $C(v)$ and $C(u)$ are also adjacent, otherwise they should not be adjacent.
- The union C of all simple configurations $C(v)$ is itself a simple configuration (i.e. there are no holes contained between configurations either).

In Figure 1.11, an arbitrary mosaic cartogram of a decomposed T is illustrated.

1.4 Results and organization

While the original solution for generating mosaic drawings by Cano et al. [2] attempts to create mosaic cartograms, it does not guarantee it. In this thesis, we introduce and

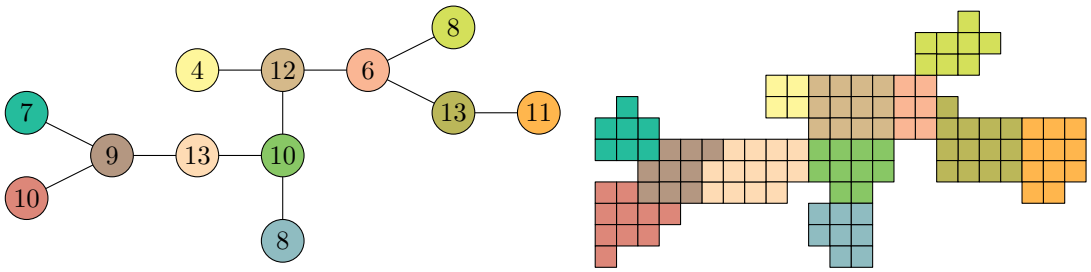


Figure 1.11: A mosaic cartogram of T.

implement a solution which can guarantee the generation of a mosaic cartogram, which assures that a grid map generated from it does not have more or less tiles than desired.

We first theorize a few principles to guarantee the construction of a mosaic cartogram, with a focus on maintaining the rules of adjacency and the number of tiles dictated by the decomposition step, by making certain assumptions and modifying the decomposition step to enforce said assumptions (Chapter 2). Secondly, we create a mosaic cartogram that is space-efficient using the previously theorized principles, after which we shape it tile by tile so that the global shape of the mosaic cartogram resembles that of the input map as close as possible (Chapter 3). Finally, we discuss the resulting shaped mosaic cartogram, identify areas of further improvement as well as identify existing limitations of our solution and proposing ways to either fix or alleviate said limitations (Chapter 4).

Chapter 2

A rule-preserving mosaic cartogram

In Chapter 1, we defined a mosaic cartogram based on a certain type of graph: a tree with nodes of at most degree 3. Furthermore, this tree was decomposed into simpler tree graphs, namely a single spine and multiple rooted binary trees. In this chapter, we discuss the construction of this tree based on the input as well as how to achieve its decomposition, and we present a number of principles to guarantee the creation of a mosaic cartogram that does not break any of its own rules based on this tree. In order to make such a guarantee, we make additional assumption about the input, regarding the minimum number of tiles certain configurations must have.

2.1 Different partitioning criteria

As discussed in Section 1.1, we modify the partitioning algorithm used by Meulemans et al. [1]. The decomposition algorithm is meant to partition a given shape into smaller shapes (or *regions*) based on salient features. It recursively divides one shape into two smaller shapes by introducing a dividing line segment (or *cut*) through the big shape. This procedure is influenced by two parameters:

- productivity - the parameter which controls the minimum number of sites that a region should contain.
- dilation - the parameter which controls the minimum size of a cut based on the Euclidean length of the resulting region.

In order for a cut to be made, its productivity and dilation is computed and if both parameters are above the given thresholds, these checks pass and the cut is accepted. Once a cut is accepted, the productivity and dilation are once again computed for any subsequent cut, taking into account the Euclidean length of the previous regions' perimeter, and only accepted if they pass both criteria.

The first change we make to the partitioning algorithm pertains to the minimum dilation check. In the original implementation, the dilation check is re-computed per each cut according to the regions introduced by previous cuts. However, we want to ensure that there are enough cuts to be considered such that our wanted property of a binary tree dual graph is met. Thus, we compute the dilation for each considered cut only once, based on the entire input shape, such that accepted cuts do not reduce the number of subsequent cuts.

We add a new check for accepting a partitioning cut: make sure that any resulting region after the cut does not have more than 3 neighboring regions. If the cut fails this check, but passes the productivity and dilation checks, this cut is not accepted but is saved for later consideration, in case that this cut becomes viable after a subsequent cut. After all cuts are tried, the saved cuts are checked again and if at least one cut becomes valid, the whole process starts again until no further cuts can be introduced without failing the check. Additionally, we modify the dilation check such that the dilation ratio is not re-computed after accepting each accepted cut. The differences between the partitioning of Meulemans et al. [1] and our partitioning can be seen in Figure 2.1.

The aim of our partitioning algorithm is to create a region per each salient feature of the map, with as few regions as possible, while also ensuring that no region has more than 3 neighbors.

2.2 Tree datastructure

The partition discussed in the previous section, consisting of adjacent regions, needs to be converted into a tree. We start by defining a *border-edge* as the line segment between any two adjacent regions, which is of equal size for both regions.

In order to create the tree T for such a partitioning, we construct a set of nodes V and

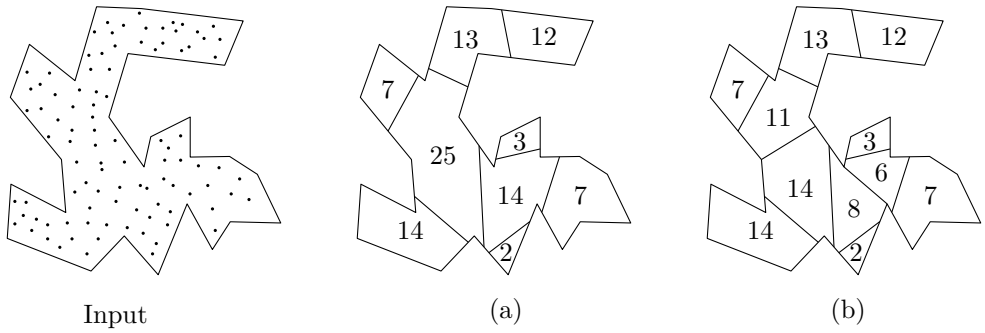


Figure 2.1: A side-by-side comparison between partitioning algorithms, given the same Input. Partition (a) accepts fewer cuts due to re-computing dilation per each accepted cut, and also allows for regions with more than 3 neighbors. Partition (b) takes more cuts, such that we do not have regions with more than 3 neighbors.

a set of edges E in the following way:

1. A node v is created for each region inside the partitioning. Additionally, v is augmented with the following data, extracted from the region: the region's centroid (or center of mass) R_v and the length of the border-edge $l_{(v,u)}$, shared between v and an adjacent region's node u.
2. An edge $e = (v, u)$ is created for each pair of nodes v, u if and only if the regions that are represented by v and u are adjacent.

Once computed and augmented, tree T is ready to be decomposed into a spine and binary trees. An example of such a tree can be seen in Figure 2.2.

2.2.1 Choosing the spine

A spine must be chosen before the binary trees are constructed. While it is entirely possible to algorithmically compute a subset V of tree T's nodes, with $h, t \in V$ such that a spine $S(V, h, t)$ can be constructed, subset V is instead manually selected, since we do not know how to compute it efficiently. The criteria for what would constitute a "good" spine are hard to determine and are outside the scope of this thesis (an example of a "good" and "bad" spine can be seen in Figure 2.3). However, we postulate the idea that a "good" spine might constitute a subset V based on the following criteria:

- the sum of all $deg(v)$ values is maximized, where $v \in V$, i.e. the chosen spine contains the maximal number of degree 3 nodes.
- the sum of edge distances $\sum_{u \in V} d(v, u)$ is minimized, where $v \in V, u \notin V$, i.e. the chosen nodes are as "central" as possible.
- based on the values of the nodes' R value (their region's centroid) the chosen nodes are as close to co-linear as possible.

Once such a subset V of T's nodes has been made, the nodes in V form a spine $S(V, h, t)$, and are referred to as *spinal nodes*. With h being the root of the spine, each spinal node v is augmented with its own node type (parent spinal node or child spinal node). Thus,

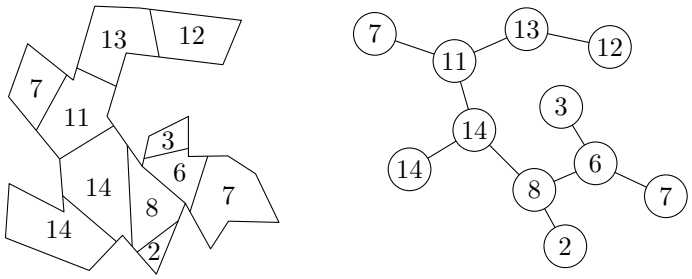


Figure 2.2: The conversion of a partitioning to a tree.

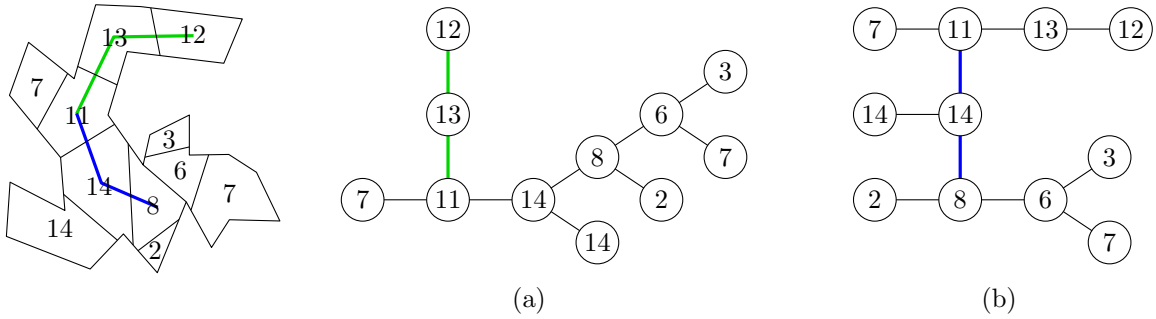


Figure 2.3: An example of a tree constructed with a "bad" spine highlighted in green (a) and tree constructed with a "good" spine highlighted in blue (b).

if a node u is adjacent to a spinal node v, it is known if u either a parent spinal node, a child spinal node or a binary tree node.

2.2.2 Creating the binary trees

After the spine has been created, removing the spinal nodes of T and the edges that connect to them leaves us with a set of disconnected subgraphs. Our aim is to converted them into rooted binary trees, the root of each tree being the node that was originally adjacent to a spinal node. We refer to a node that is part of a binary tree as a *non-spinal node*.

We construct a binary tree as follows: for each non-spinal node, starting from the root, we check how many neighboring nodes it has. If any of the neighboring nodes are neither a spinal node or an already processed node (i.e. a parent), they are assigned to be the current node's children, making the current node a parent node. The decision regarding which nodes are considered "left" or "right" children is made based on the partition's embedding, which is discussed in more detail in Section 2.2.3.

After building the binary tree, we now augment each of its nodes n with its corresponding k_n value, that is, the number of degree 3 nodes (2 children and a parent) contained by the subtree rooted at n. For each node, starting from the root, if n's degree is 3, k_n is incremented by 1, and if the node has any children c, k_n is incremented by the values k_c, which are calculated recursively. A figure of the resulting binary trees given a partition can be seen in Figure 2.4.

2.2.3 Embedding preservation

We want our tree to be structured such that given partition's dual graph embedding is preserved. In order to preserve this embedding, we make use of the nodes' stored centroid values, which are a reflection of the partition's dual graph embedding.

Since the spine is a path graph, the embedding for spinal nodes is already preserved because a spinal node only connects to at most 2 other spinal nodes. This can be seen

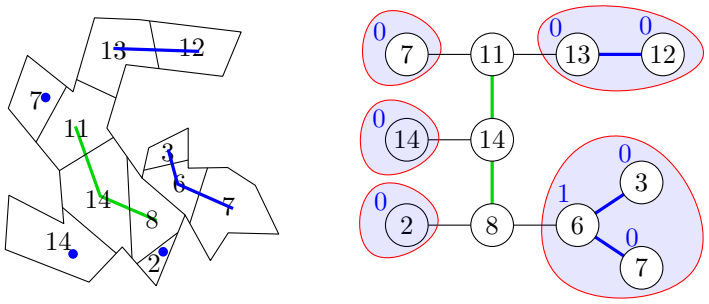


Figure 2.4: A partition and its equivalent tree showcasing the spine (blue) and binary trees (blue) as well as the augmented k value for each non-spinal node (also in blue).

in Figure 2.5

The embedding given by our partition's dual graph is not inherently preserved in our binary trees. Figure 2.6 shows an incorrect and correct embedding representation of a given partition's binary tree. In order to preserve it, the clockwards order of the partition must also be preserved in our binary tree, in the form of a "left" child node and a "right" child node.

Traditionally, when visualizing binary trees, "left" and "right" children of a node are assigned in relation to their parent node's visual placement. The parent node is arbitrarily placed, and its children are placed below it, such that the left child is placed westward in relation to the parent node, while the right child is placed eastward in relation to the parent node.

We want to preserve the embedding using this convention. However, in our tree T, not all nodes are visually placed that way, so determining which node should be "left" or "right" can become tricky. However, using the established convention, we induce some criteria that can produce a consistent labeling of "left" and "right" children in relation to a parent node.

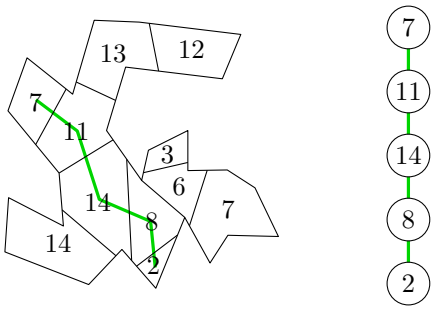


Figure 2.5: A partition with an arbitrary spine (green) and its equivalent tree. The embedding is clearly preserved.

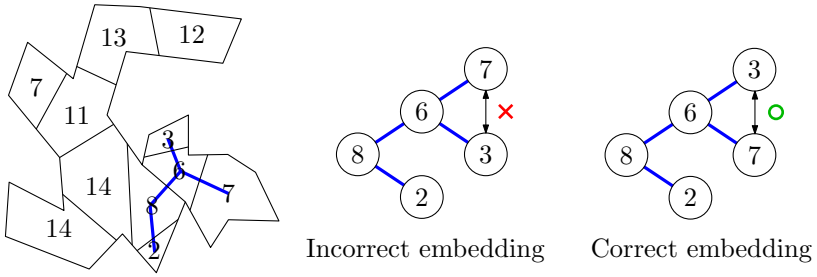


Figure 2.6: A partition with an arbitrary binary tree (blue) and its two equivalent trees of different embeddings.

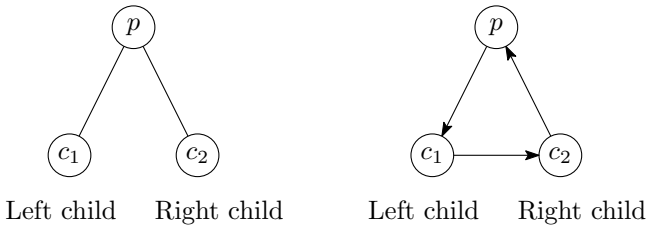


Figure 2.7: The "left" and "right" child convention for binary trees, and its counter clockwise relation.

We observe that in the current convention, the parent node, left child and right child are in counter-clockwise order, showcased in Figure 2.7. Using this observation, we label a child node either "left" or "right" in the following manner: Assuming a node p is the parent of two children c_1 and c_2, we compute the signed area of the triangle formed by the points R_p, R_{c_1} and R_{c_2}. If the signed area is positive, we know that the mentioned points are distributed in a counter-clockwise manner in relation to p, and assign c_1 and c_2 to be the "left" and "right" children of p respectively. Otherwise, c_1 and c_2 are assigned to be the "left" and "right" children of p respectively. Using this criteria, the partition's dual graph embedding within binary trees is preserved, as can be seen in Figure 2.8.

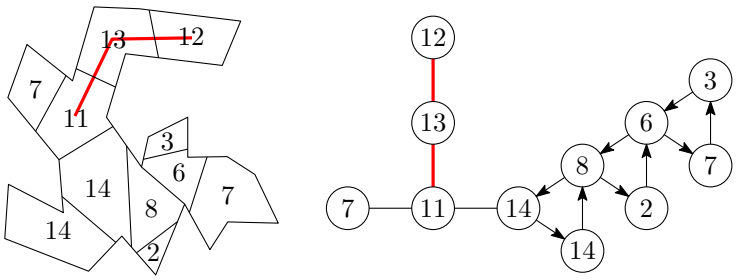


Figure 2.8: The binary tree children convention applied for a given partition with an arbitrary spine (red).

Lastly, the way that the roots of the binary trees connect with a spinal node also induce a certain embedding. In this thesis, to produce the partition's dual graph embedding, the spine is constructed such that the spinal nodes are placed vertically, thus the embedding criteria for determining which binary tree root is placed "left" or "right" in relation to a spinal node consists of comparing the X-axis value of both tree roots' centroids, assigning the "left" placement to the node with the smallest value and the "right" placement to the node with the biggest value.

Thus, we are able to preserve the partition's dual graph embedding in our tree datastructure T.

2.3 Steps for creating a mosaic cartogram

Once our tree datastructure T has been decomposed into a spine and a set of binary trees, we are almost ready to create a mosaic cartogram of it. We start by defining the two basic configuration types.

We define a vertical/horizontal slab to be a specific type of configuration, where the tiles are edge-connected in sequence, along one spatial dimension, either horizontal or vertical. Additionally, we define an L-slab as a configuration whose tiles arrangement can be decomposed into one horizontal slab and one vertical slab that form an L shape such that no tile within this configuration is edge-connected with more than 2 tiles. These types of configurations can be seen in Figure 2.9.

In order to construct a basic mosaic cartogram of T using only these two configuration types, we propose the following methodology:

1. For each spinal node n in spine $S(V_S, h, t)$, starting with h, $C(n)$ is a vertical slab.
2. For each node n of a binary tree $B(V_u, u)$, starting with r, $C(n)$ is either a horizontal slab if $deg(n) < 3$ or an L-slab if $deg(n) = 3$.

The first part of creating a mosaic cartogram of T is by chaining each spinal node's vertical slab configuration according to the spine adjacencies such that the union of these configurations is itself a vertical slab. We can trivially see that the correct adjacencies are maintained between configurations, and each configuration's tile number is precise, since the vertical slab can expands upwards or downwards infinitely (as seen in Figure 2.10).

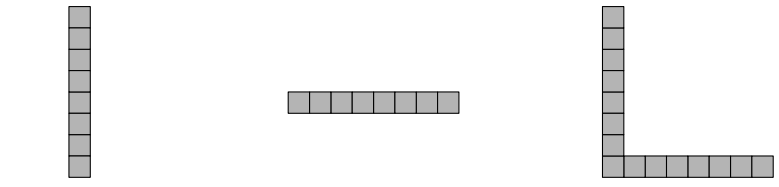


Figure 2.9: Three types of configurations.

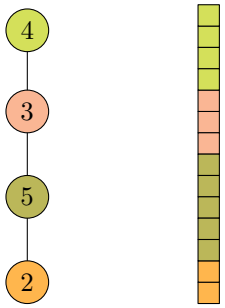


Figure 2.10: The mosaic cartogram of an arbitrary spine, composed of a single vertical slab.

The second part of creating the mosaic cartogram of T consists of arranging the configurations of the non-spinal nodes, which all belong to a binary tree. We do this per each binary tree. The configuration of a non-spinal node is either a horizontal slab or an L-slab, and each node's configuration is placed according to their tree adjacencies and embedding. An example of this can be seen in Figure 2.11. In order to preserve the correct adjacencies and number of tiles for each configuration, we make an additional assumption on T and prove the following lemma:

Lemma 1. *Given a rooted, augmented binary tree B with k degree 3 nodes and an embedding, and assuming that the sq_v value of every node v of T is at least $sq_{min} = 2(k+1)$, a mosaic cartogram of B can be constructed within $2k+1$ vertical space on a square tiling $\mathcal{T}$.*

2.3.1 Proof

Induction hypothesis: given a rooted, augmented binary tree B with k degree 3 nodes and an embedding, and assuming that the sq_v value of every node v of T is at least $sq_{min} = 2(k+1)$, a mosaic cartogram of B can be constructed within $2k+1$ vertical space on a square tiling $\mathcal{T}$.

Base case $k = 0$ The vertical space available is $2 \cdot 0 + 1 = 1$. The tree consists of nodes connected in sequence. Thus, the mosaic cartogram takes the shape of a horizontal slab, occupying one row of vertical space (Figure 2.12). Thus, the IH trivially holds.

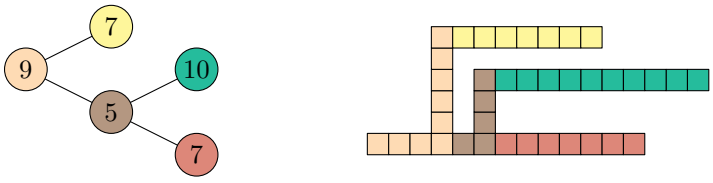


Figure 2.11: The mosaic cartogram of a binary tree, composed of L-slabs and horizontal slabs.

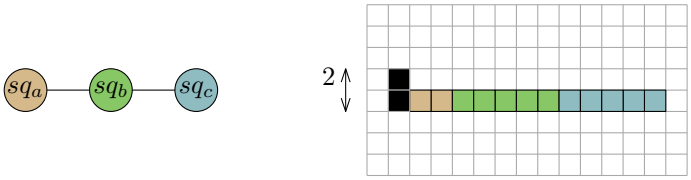


Figure 2.12: The mosaic cartogram of a tree with no degree 3 nodes, composed of a horizontal slab, with $sq_a = 2, sq_b = 5, sq_c = 5$.

Inductive step case $k > 0$ We wish to prove that, for the given tree B with k degree 3 nodes and embedding, there is $2k + 1$ available vertical space to construct its mosaic cartogram.

The given tree B's mosaic cartogram is constructed as follows: starting from the root and until a degree 3 node is encountered, every degree 2 and degree 1 nodes have their configurations constructed as chained horizontal slabs in any of the $2k + 1$ rows of available space. Once a degree 3 node is reached, its configuration is constructed partly as a horizontal slab either in any of the $2k + 1$ rows of available space, or chained to an existing horizontal slab if it exists. Then, the rest of its configuration shall extend up and/or down, forming a vertical slab of $2k$ tiles within the confines of the given $2k + 1$ vertical space, forming an L-slab. This construction is possible since the minimum number of tiles for any given node is $sq_{min} = 2(k + 1)$, more than the available vertical space, ensuring that the vertical slab subset of its configuration can be constructed.

Once a node p of degree 3 with k_p degree 3 nodes in its subtree has its configuration constructed, two subtrees B_l (subtree rooted on p's left child, given by B's embedding) and B_r (subtree rooted on the p's right child, given by B's embedding) need to be have their mosaic cartograms constructed. B_l contains k_l degree 3 nodes, and B_r contains k_r degree 3 nodes, with $k_l + k_r = k_p - 1$.

Continuous space is reserved for two subtrees B_l and B_r according to the number of degree 3 nodes each tree has, and relative to their embedding. Additionally, one row of space between them is needed to ensure adjacency separation. This is showcased in Figure 2.13.

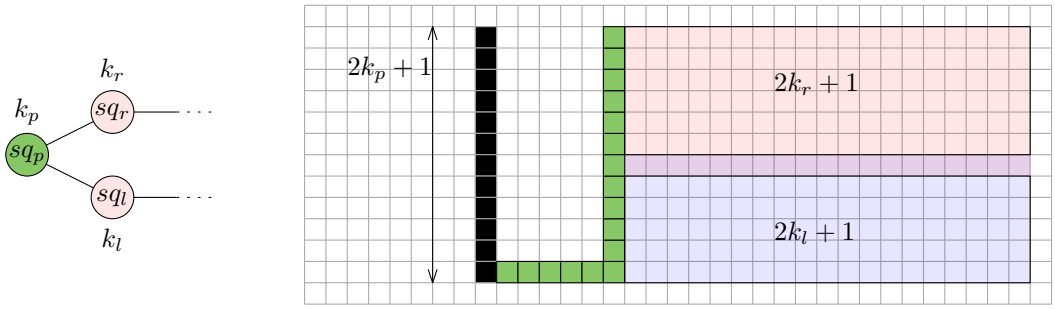


Figure 2.13: Space allocation for B_l and B_r, according to their respective k value.

Given subtree B_l with k_l degree 3 nodes, $2k_l + 1$ vertical space is required, and given subtree B_r with k_r degree 3 nodes, $2k_r + 1$ vertical space is required. With the separation row between them, the total vertical space used is $2(k_l + k_r + 1) + 1$.

Since we know that there are $k_p - 1$ total degree 3 nodes left in both subtrees, this means that $k_l + k_r = k_p - 1$, meaning $k_p = k_l + k_r + 1$. Thus, to construct B's mosaic cartogram, the space needed is $2(k_l + k_r + 1) + 1 = 2k_p + 1$.

Thus, Lemma 1 holds.

2.4 A basic mosaic cartogram

With the added assumption that every binary tree B with k_B degree 3 nodes in a decomposition of tree T nodes has a lower bound of $sq_{min}(B) = 2(k_B + 1)$ such that every node v in B has its $sq_v \geq sq_{min}$, it is possible to create a basic mosaic cartogram of any such tree T, with emphasis on maintaining correct adjacencies and the specified number of tiles.

Similarly to the problem of creating a "good" spine, enforcing this added assumption is done manually by instructing the partition algorithm to create regions with a chosen minimal number of sites (through its productivity parameter) such that the assumption is met for any spine. Thus, we give an arbitrary value bigger than $\max_{B \in T}(sq_{min}(B))$ as a productivity parameter for the partitioning algorithm, enforcing a lower bound sq_{min} value for each node v's sq_v value, where v is a node in T. The reasoning behind this is that minimizing the assumption's $sq_{min}(B)$ value per each binary tree B is a hard problem of its own and is outside the scope of this thesis, thus we use a value that works for all binary trees. However, we intuit that the minimization of $sq_{min}(B)$ is tied to both the partitioning algorithm and the automatic spine creation problem discussed in Section 2.2.1.

Figure 2.14 contains an example of an arbitrary tree T that meets our discussed assumptions.

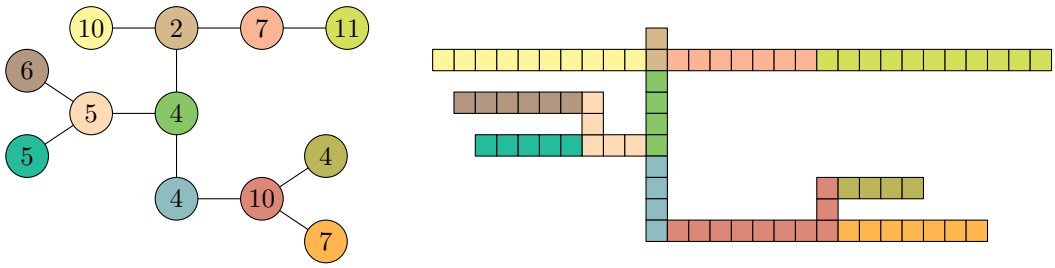


Figure 2.14: A mosaic cartogram of an arbitrary tree T.

Chapter 3

Shaping the mosaic cartogram

In this chapter, we build upon the theoretical mosaic cartogram discussed in Chapter 2 to create a mosaic cartogram that resembles a given partition's shape as much as possible.

As discussed in Section 2.1, a partition's goal is to capture salient features of a shape with the least number of partitioned regions, and combined with a minimum productivity for our partition, in practice, the number of sites per region is not always as small as our productivity parameter. If we exclusively use Chapter 2's mosaic cartogram principles, we notice that the resulting mosaic cartogram expands drastically both vertically and horizontally, not making efficient use of the available space. The shape of the mosaic cartogram would not resemble the given partition in any form either. Thus, we identify two steps to address these issues: first, making a space-efficient, compact version of the basic mosaic cartogram, and second, using this mosaic cartogram as a starting point and shaping it such that the final mosaic cartogram resembles the partition as much as possible, remaining a valid mosaic cartogram throughout with each shaping step.

3.1 Constructing a compact mosaic cartogram

For a partitioning that contains regions with a large number of sites, the basic mosaic cartogram excessively extends vertically and horizontally, leaving a lot of unused space, as seen in Figure 3.1. In order to address this, we propose an algorithm that aims to create a compacted mosaic cartogram while still maintaining the mosaic cartogram rules, under the same assumptions as the basic mosaic cartogram.

We preserve the same idea that the basic mosaic cartogram has: spinal nodes are placed vertically and non-spinal nodes are placed horizontally. Each spinal node v's height is denoted by $height_v$, and each binary tree B rooted at r (which is adjacent to v) has its height maximally bound by $height_v$. In order to enforce Lemma 1 each configuration of $C(u)$ has to have a height bound between 1 and $height_v$, where $height_v \geq sq_{min}$.

For each node v, we define $L_{(v,u)}$ to be a subset of $C(v)$ such that it contains all tiles

Figure 3.1: An arbitrary tree T and its basic mosaic cartogram, we can see that it occupies a lot of horizontal space.

incident to $C(u)$ (and since the adjacency rule is mandatory, $L_{(v,u)}$ must contain at least one tile), and is either a vertical slab (if u is a non-spinal node) or a horizontal slab (if u is a spinal node). Additionally, $\bigcap L_{(v,u)} = \emptyset$. We denote $|L_{(v,u)}|$ to represent the number of tiles that $L_{(v,u)}$ contains. This is exemplified in Figure 3.2.

We aim to compact the nodes using different criteria, based on if a node is a spinal or non-spinal. We first apply a compaction of spinal nodes, after which we apply a compaction of non-spinal nodes. These procedures are discussed in the upcoming sections.

3.1.1 Proportional compaction of spinal nodes based on polygon area

To compact a spinal node v, we implemented an algorithm that attempts to arrange its configuration such that it resembles the shape of a specific polygon, based on v's adjacent nodes, detailed below.

Given a spinal node v and one of its neighbor node u, we denote $l_{(v,u)}$ to be the length of the border-edge discussed in Section 2.2). For each node v, there at most 3 such border-edge lengths, since v can have at most 3 neighboring nodes. Thus, based on these lengths, we are able to compute the area of a polygon with at most 4 edges.

For each spinal node v, we want to make its configuration $C(v)$ such that each $|L_{(v,u)}|$ is proportional to the corresponding border-edge length $l_{(v,u)}$ in the partition. In general this cannot be satisfied exactly, due to rounding errors, so in the remainder of this section

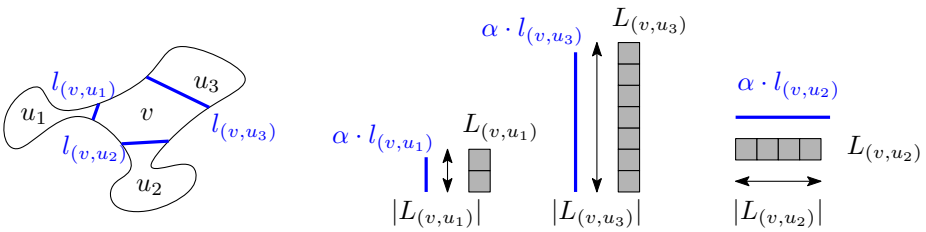


Figure 3.2: An arbitrary partition with three border-edges with length $l_{(v,u_i)}$, where where $0 < i \leq 3$, v, u_2 are spinal nodes and u_1, u_3 are a non-spinal nodes, and their corresponding $L_{(v,u_i)}$ sets given an arbitrary proportionality constant α.

we describe how this proportionality is achieved with the added constraint that $C(v)$ resembles the shape of a polygon to achieve some level of compaction.

Take a spinal node v with u_i as a neighbor, where $0 < i \leq 3$. Proportionality with $l_{(v,u_i)}$ implies that $\frac{|L_{(v,u_1)}|}{l_{(v,u_1)}} = \frac{|L_{(v,u_2)}|}{l_{(v,u_2)}} = \frac{|L_{(v,u_3)}|}{l_{(v,u_3)}}$. Assuming that sq_v is the area of a known polygon, we can determine a proportionality constant α such that the area of this polygon is computed using the lengths $\alpha \cdot l_{(v,u_i)}$. We then compute $|L_{(v,u_i)}| = \lfloor \alpha \cdot l_{(v,u_i)} \rceil$.

The shape of the polygon depends on depending on the following cases:

- v has only one neighbor u_1: since there is only one slab $L_{(v,u_1)}$, the polygon in this case is a rectangle, as seen in Figure 3.3.
- v has two neighbors u_1 and u_2: if u_1 and $u2$ are both either spinal nodes or binary tree roots, the polygon in this case is a right trapezoid. if u_1 and u_2 are different type of nodes, the polygon is a right triangle, as seen in Figure 3.4.
- v has three neighbors u_1, u_2 and u_3: this implies that at least one of them is a non-spinal node. The polygon in this case is a trapezoid, as seen in Figure 3.5.

We enforce the following relation between any two neighboring spinal nodes v and u: $|L_{(v,u)}| = |L_{(u,v)}|$. This is done in order to maintain an equal number of incident tiles in both $C(v)$ and $C(u)$ similar to how the border-edge is of same length between the regions of v and u. If, after computing the polygons for each node $|L_{(v,u)}| \neq |L_{(u,v)}|$ (As seen in Figure 3.6), we make it so that $|L_{(v,u)}| = |L_{(u,v)}| = \min(|L_{(v,u)}|, |L_{(u,v)}|)$ (the number of incident tiles in both nodes match in number) and recompute either $L_{(v,u)}$ or $L_{(u,v)}$ according to v or u's border-edge length proportionality. Enforcing this relation implies that any change snowballs into the all other spinal nodes. This is exemplified in Figure 3.7.

If a spinal node v had its $|L_{(v,u_i)}|$ changed to match $|L_{(u_i,v)}|$, where $i \neq j \neq y$, we then recompute $|L_{(v,u_j)}|$ and $|L_{(v,u_y)}|$ based on proportionality with $l_{(v,u_i)}$, $l_{(v,u_j)}$ and $l_{(v,u_y)}$, which implies that

$$\frac{|L_{(v,u_i)}|}{l_{(v,u_i)}} = \frac{|L_{(v,u_j)}|}{l_{(v,u_j)}} = \frac{|L_{(v,u_y)}|}{l_{(v,u_y)}}.$$

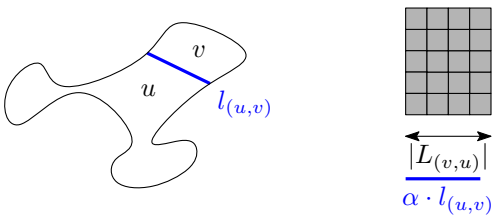


Figure 3.3: An arbitrary partition with a spinal node v and a non-spinal node u with one border-edge between them of length $l_{(v,u)}$. α is computed such that $C(v)$ resembles a rectangle with an edge of length $|L_{(v,u)}|$.

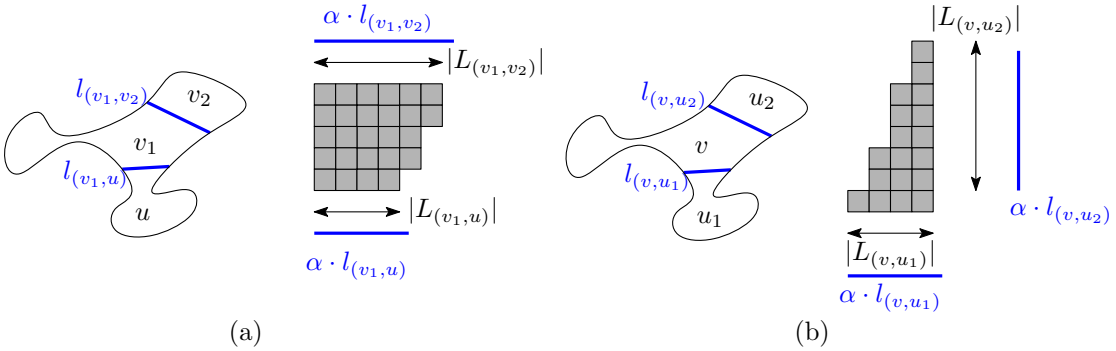


Figure 3.4: Figure (a) shows an arbitrary partition with two spinal nodes v_1 and v_2 one non-spinal node u with their corresponding border-edge lengths. α is computed such that $C(v_1)$ resembles a right trapezoid with its two parallel edges of length $|L_{(v_1,v_2)}|$ and $|L_{(v_1,u)}|$.
Figure (b) shows an arbitrary partition with a spinal node v and two non-spinal nodes u_1 and u_2 with their corresponding border-edge lengths. α is computed such that $C(v)$ resembles a right triangle with catheti of length $|L_{(v,u_1)}|$ and $|L_{(v,u_2)}| + 1$.

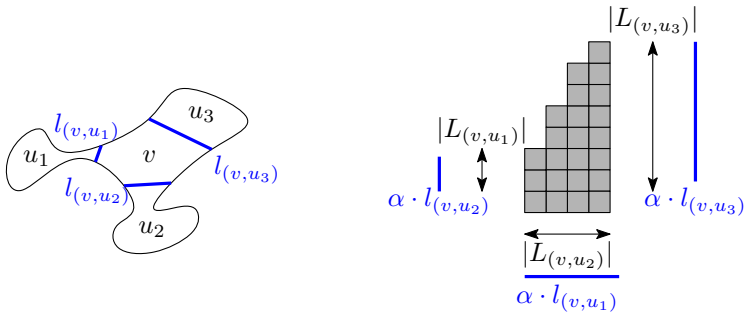


Figure 3.5: An arbitrary partition with a spinal node v and three nodes u_1, u_2 and u_3 with their corresponding border-edge lengths. α is computed such that $C(v)$ resembles a right trapezoid with edges of length $|L_{(v,u_1)}|$, $|L_{(v,u_2)}|$ and $|L_{(v,u_3)}|$, performing an addition of 1 to this length if u_i is a non-spinal node, $0 < i \leq 3$.

Thus,

$$|L_{(v,u_j)}| = \lfloor \frac{|L_{(v,u_i)}|}{l_{(v,u_i)}} \cdot l_{(v,u_j)} \rceil$$

and

$$|L_{(v,u_y)}| = \lfloor \frac{|L_{(v,u_i)}|}{l_{(v,u_i)}} \cdot l_{(v,u_y)} \rceil.$$

This algorithm is applied recursively starting on head node of the spine and ending on the tail node of the spine. After finishing, the configurations of all spinal nodes are compacted to a polygonal figure of at least the number of tiles in their configuration,

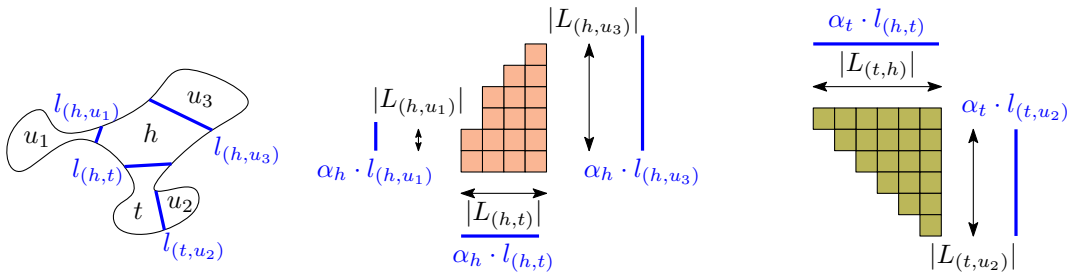


Figure 3.6: An arbitrary partition with two spinal nodes h and t and three nodes u_1, u_2 and u_3 with their corresponding border-edge lengths. h and t's proportionality constants α_h and α_t are computed such that the shapes of $C(h)$ and $C(t)$ are corresponding to their appropriate case.

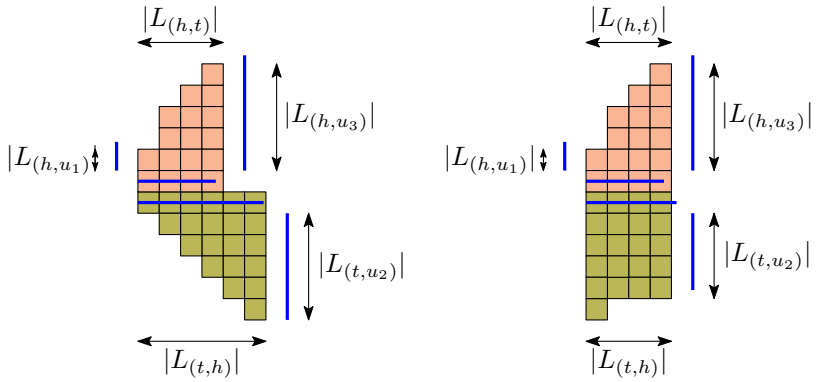


Figure 3.7: Given the partition in Figure 3.6, $|L_{(h,t)}| \neq |L_{(t,h)}|$. Thus we make $|L_{(t,h)}|$ equal to $|L_{(h,t)}|$, changing the proportionality constant α_t to α'_t and as a consequence, recompute $L_{(h,t)}$ according to α'_t.

while maintaining proportionality with the partition border-edges.

Given any spinal node v and a neighboring non-spinal node u, where u is the root of binary tree B, it may happen that $L_{(v,u)}$ has less tiles than $sq_{min}(B)$, in which case we revert the entire mosaic cartogram of the spine to the basic mosaic cartogram discussed in Section 2.3 (seen in Figure 2.10).

After applying this algorithm, we proceed with compacting the non-spinal nodes i the next section.

3.1.2 Proportional compaction of non-spinal nodes based on border-edge lengths

To compact a non-spinal node v neighboring a spinal node s, we implemented an algorithm that attempts to arrange its configuration such that its tiles are contained within $height_s$. Given a non-spinal node v and a neighboring non-spinal node u, we denote the

border-edge length as $l_{(v,u)}$.

Take a non-spinal node v with u_i as a non-spinal neighbor, where $0 < i \leq 3$. Proportionality with $l_{(v,u_i)}$ implies that $\frac{|L_{(v,u_1)}|}{l_{(v,u_1)}} = \frac{|L_{(v,u_2)}|}{l_{(v,u_2)}} = \frac{|L_{(v,u_3)}|}{l_{(v,u_3)}}$. Knowing that we have a total of sq_v tiles and that given a height bound $\min(height_s, sq_v)$, we can determine a proportionality constant α such that $\alpha \cdot \max_{0<i\leq 3} l_{(v,u_i)} = H$. We then compute $|L_{(v,u_i)}| = \lfloor \alpha \cdot l_{(v,u_i)} \rceil$.

Similar to the spinal nodes, we enforce the following relation between any two neighboring non-spinal nodes v and u: $|L_{(v,u)}| = |L_{(u,v)|}$. If, after computing $L_{(v,u)}$ and $L_{(u,v)}$, $|L_{(v,u)}| \neq |L_{(m,v)}|$, we make it so that $|L_{(v,u)}| = |L_{(m,v)}| = \min(|L_{(v,u)}|, |L_{(u,v)}|)$ and recompute either $L_{(v,u)}$ or $L_{(u,v)}$ according to v or u's border-edge length proportionality. Enforcing this relation implies that any change snowballs into the all other non-spinal nodes (unless stated otherwise). This is illustrated in Figure 3.8.

If a non-spinal node v had its $|L_{(v,u_i)}|$ changed to match $|L_{(u_i,v)}|$, where $i \neq j \neq y$, we then recompute $|L_{(v,u_j)}|$ and $|L_{(v,u_y)}|$ based on proportionality with $l_{(v,u_i)}$, $l_{(v,u_j)}$ and $l_{(v,u_y)}$, which implies that

$$\frac{|L_{(v,u_i)}|}{l_{(v,u_i)}} = \frac{|L_{(v,u_j)}|}{l_{(v,u_j)}} = \frac{|L_{(v,u_y)}|}{l_{(v,u_y)}}.$$

Thus,

$$|L_{(v,u_j)}| = \left\lfloor \frac{|L_{(v,u_i)}|}{l_{(v,u_i)}} \cdot l_{(v,u_j)} \right\rceil$$

and

$$|L_{(v,u_y)}| = \left\lfloor \frac{|L_{(v,u_i)}|}{l_{(v,u_i)}} \cdot l_{(v,u_y)} \right\rceil.$$

If a non-spinal node v has two children c_1 and c_2 and a parent p, the above relation is not always enforceable, because with each degree 3 node, the two subtrees B_1 and B_2

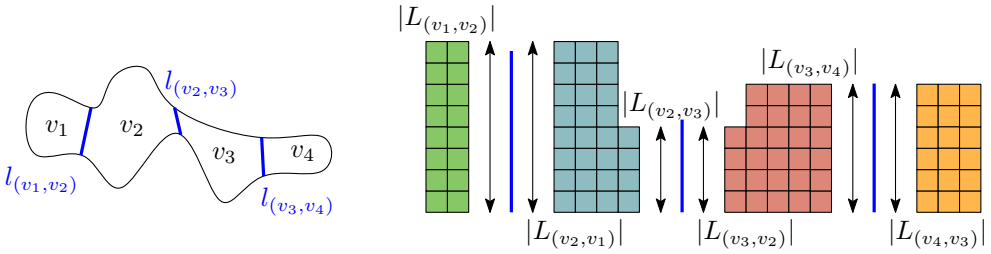


Figure 3.8: An arbitrary partition with four non-spinal nodes v_i, $0 < i \leq 3$ with their corresponding border-edge lengths. The blue segments are scaled versions of the border-edge lengths such that the relation $|L(v_i, v_j)| = |L(v_j, v_i)|$ holds, where $0 < i, j \leq 3$ and $i \neq j$.

rooted at c_1 and c_2 respectively, need a different minimum height than $2k_n + 1$, namely $2k_{c_1}$ and $2k_{c_2}$. The maximum subtree heights $height_{B_{c_1}}$ and $height_{(B_{c_2})}$ are dictated by the number of tiles in $L_{(v,c_1)}$ and $L_{(v,c_2)}$ respectively. The maximum height of node v's configuration is dictated by the number of tiles in $L_{(v,p)}$. This is illustrated in Figure 3.9.

However, there are two cases where trying to enforce that $|L_{(v,c_1)}| = |L_{(c_1,v)}|$ and $|L_{(v,c_2)}| = |L_{(c_2,v)}|$ would not work:

- $|L_{(v,c_1)}| + |L_{(v,c_2)}| > |L_{(v,p)}| - 1$. The computed height values exceed v's maximum configuration height minus one vertical space needed to maintain adjancencies as per Lemma 1, as seen in Figure 3.10.
- $|L_{(v,c)}| < 2k_c + 1$, where $c \in \{c_1, c_2\}$. One or more computed height values are lower than the minimum height needed for subtree B_c.

In these cases, $|L_{(v,p)}|$, $L_{(v,c_1)}$ and $L_{(v,c_2)}$ are recomputed such that $|L_{(v,c_1)}| + |L_{(v,c_2)}| \leq |L_{(v,p)}| - 1$, where $|L_{(v,p)}| \geq 2k_v + 1$, $|L_{(v,c_1)}| \geq 2k_{c_1} + 1$ and $|L_{(v,c_2)}| \geq 2k_{c_2} + 1$. This is done by computing

$$|L_{(v,p)}| = \left\lfloor \min(height_s, \frac{sq_v}{2} \right\rceil$$

,

$$|L_{(v,c_1)}| = \left\lfloor (|L_{(v,p)}| - 1) \cdot \frac{k_{c_1} + 1}{k_v + 1} \right\rceil$$

and

$$|L_{(v,c_2)}| = \left\lfloor (|L_{(v,p)}| - 1) \cdot \frac{k_{c_2} + 1}{k_v + 1} \right\rceil .$$

The values of $|L_{(v,p)}|$, $|L_{(v,c_1)}|$ and $|L_{(v,c_2)}|$ are then excluded from the enforcement of the property that $|L_{(v,p)}| = |L_{(p,v)}|$, $|L_{(v,c_1)}| = |L_{(c_1,v)}|$ and $|L_{(v,c_2)}| = |L_{(c_2,v)}|$. This can be seen in Figure 3.11.

Once all $L_{(v,u)}$ have been calculated for each pair of neighboring non-spinal nodes v and u, there can be an excess of tiles that do not belong in any $L_{(v,u)}$ subset. These extra tiles are placed between the $L_{(v,u)}$ slabs, within a height bound of $\max_{u \in U} |L_{(v,u)}|$ (where U is the set of non-spinal neighbors of v), preventing the addition of holes. Ideally, the

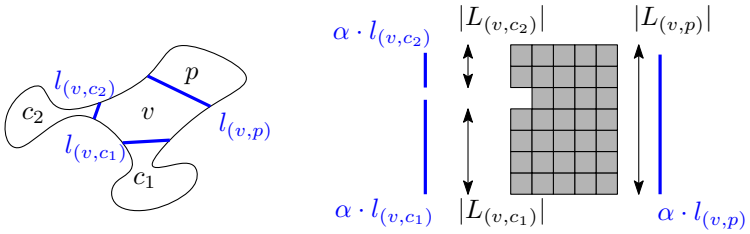


Figure 3.9: An arbitrary partition with four non-spinal nodes, showcasing v's configuration, with p as its parent node, matching border-edge proportionality.

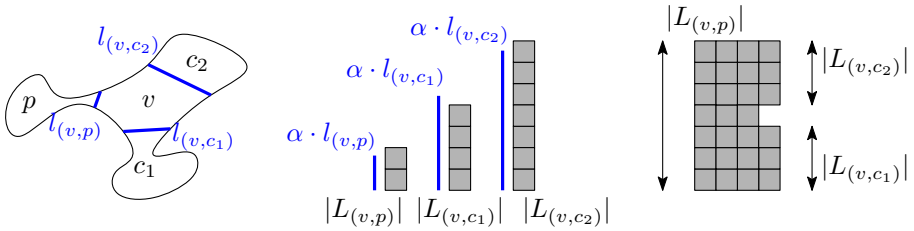


Figure 3.10: An arbitrary partition with four non-spinal nodes, showcasing v's configuration, with p as its parent node, in the case that the border-edge proportionality cannot be enforced.

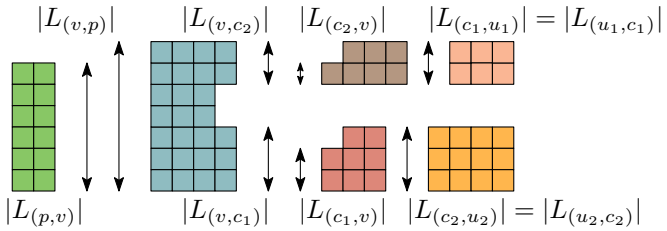


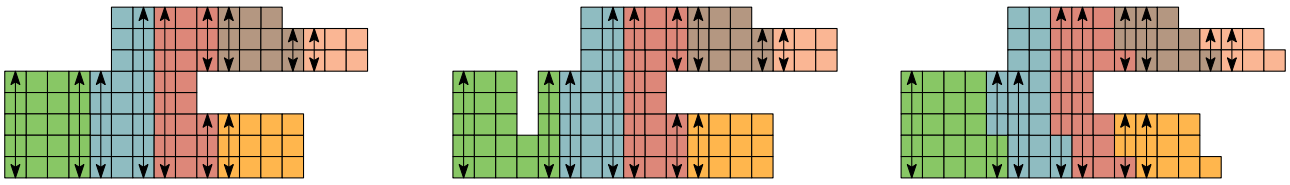
Figure 3.11: A non-spinal node v with a parent p and two children c_1 and c_2 where the relation $L_{(v,w)} = L_{(w,v)}$, with $w \in \{p, c_1, c_2\}$, is not enforced on its configuration $C(v)$ (blue).

shape of each individual configuration resemble a rectangle (Figure 3.12 (a)), but there can be excess tiles that would prevent this (Figure 3.12 (b)), thus we place the excess tiles such that they are "propagated" until a leaf node is reached (Figure 3.12 (c)).

This procedure is applied for all non-spinal nodes of T which are adjacent to a spinal node.

3.1.3 A compacted mosaic cartogram

Given all procedures mentioned above, a potential compaction of an arbitrary basic mosaic diagram can be seen in Figure 3.13.



(a) (b) (c)

Figure 3.12: An "ideal" configuration scenario (a), an "imperfect" scenario (b) and an improvement of the imperfect scenario (c).

3.2 Generating guiding tiles via partition shapes

The idea to use guiding shapes in order to shape a mosaic drawing is inspired by the solution of Cano et al. [2]. Once we obtained a mosaic cartogram, our next goal is to shape it such that it resembles the shape of the partitioning as much as possible. We devise an intuitive approach for this next step: create a "guide" mosaic cartogram of the partition, which does not take into account contiguity, number of tiles, lack of holes and correct adjacency rules, and attempt to change the mosaic cartogram tile-by-tile, such that no rules are broken, "molding" our mosaic cartogram to resemble the "guide" mosaic cartogram as much as possible.

The problem we encounter is that the number of tiles across regions are not proportional with the geometric areas of the regions themselves, and as such, we cannot take the naive approach where we could simply create a rasterization of the original partition. Thus, to get around this, we propose a method to create rasterizations of regions that contain an number of tiles that is roughly the same number as the wanted number of tiles.

3.2.1 Region scaling and placement

The general idea of this method is that we scale each partition region individually such that their geometric area matches the number of tiles that this region's configuration should have. After this scaling each region, the scaled regions are positioned in a logical manner such that they still resemble the original partitioning.

For each region, we first compute the area A of the region polygon. Using the desired cartogram area A_c for this region, which is equal to the number of tiles in this region's configuration, compute the scale factor $\alpha = \sqrt{\frac{A'}{A}}$ used to scale A to the desired area A', as exemplified in Figure 3.14.

Once all regions have been scaled (Figure 3.15 (b)), they have to be pieced together such that they resemble the original partitioning as much as possible. The way we do this is

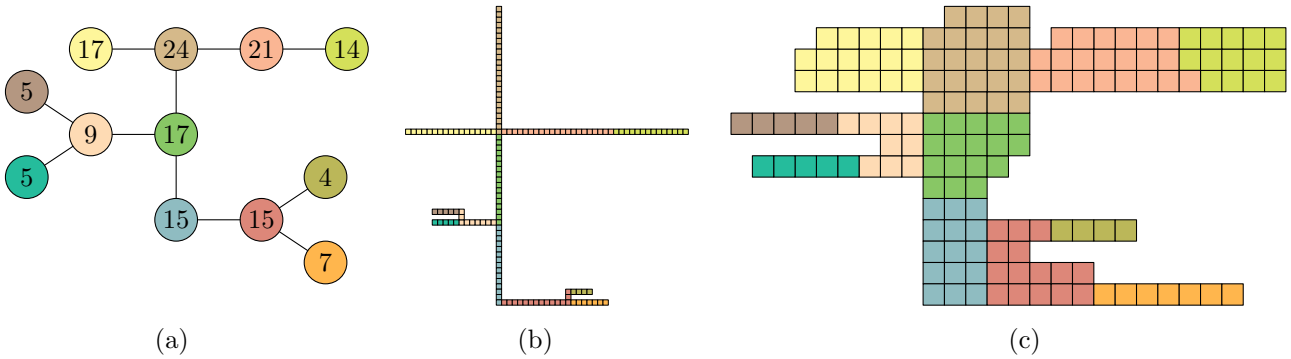


Figure 3.13: An arbitrary tree T (a) with its basic mosaic cartogram (b) and its compacted mosaic cartogram (c).

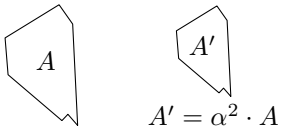


Figure 3.14: An arbitrary region with area of A scaled to A'.

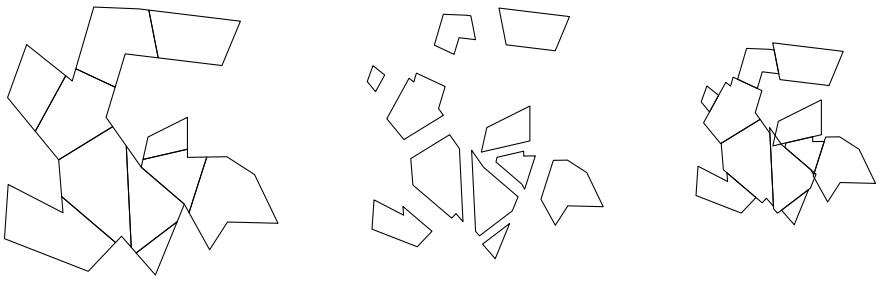
by first placing the scaled region corresponding to the head node of the spine, such that this scaled region's centroid coincides with centroid of this node's configuration.

Since each partition has been disproportionally scaled, this means that the border-edge segment previously shared between two regions are of different lengths per scaled region. However, the information of which region connects to which region is preserved by the tree datastructure, and we are able to place them according to this information. After positioning a scaled region, for each of its original neighboring regions, we place the scaled region such that the middle point of the placed region's border-edge is overlapping the middle point of the scaled neighbor region's border-edge. This results in a logically connected scaled partitioning of the original partition, which we refer to as a scaled partition. We node that, as seen in Figure 3.15, we allow for shapes overlapping each other.

3.2.2 Generating guide tiles via rasterization

Once the scaled partition has been created, the next step is to create a rasterization of each scaled region within it. For each scaled region, a rasterization is created (Figure 3.16). For brevity, we define the rasterization of a node v of T to be the rasterization of the scaled region associated with v.

We create the rasterization for the entire partition in a specific order, dictated by our tree T as follows: We first create the rasterization of the spinal nodes, starting with the spine's head node and ending with the spine's tail node. Then, we create the rasterization of non-spinal nodes of the binary tree that contains them, in breadth first order, prioritizing



(a) (b) (c)

Figure 3.15: The scaling process (b) and the positioning process (c) applied on an arbitrary partition (a).

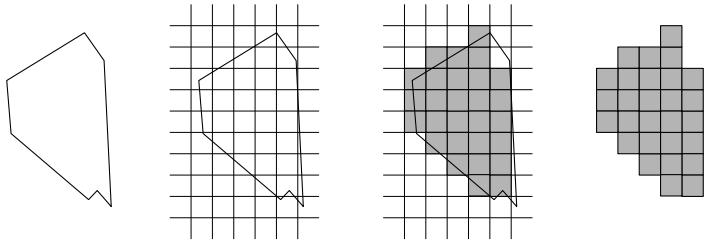


Figure 3.16: A rasterization of an arbitrary region.

the binary trees with their root that is closest to the head node of the spine. In the case that the rasterization of a scaled region would overlap a rasterization of region whose rasterization has already been made (due to scaled regions overlapping), the specific tiles that would create overlap are not placed.

The complete rasterization of the scaled partition (Figure 3.17) gives us a "guide" mosaic drawing that we want to use for the next step. Our next goal is to use this mosaic drawing as a guide in order to shape our mosaic cartogram with.

3.3 Shaping the mosaic cartogram via guide tiles

Our "guide" mosaic drawing is composed of tiles that belong to specific regions, just like our mosaic cartogram. We refer to the tiles of the "guide" mosaic drawing as *guide tiles* (tiles belonging to the same region referred to as a *guide configuration*) (Figure 3.18). From here on out, given a set of guide tiles corresponding to various guide configurations, the problem becomes a maximization problem, where the aim is to overlap as many guide tiles of a certain node's guide configuration with tiles belonging to the same node's configuration as possible, while keeping our mosaic cartogram valid with each step.

More formally, let v be a node in tree T, let $\mathcal{C}_{\mathcal{T}}(T)$ be T's mosaic cartogram and let $\mathcal{D}_{\mathcal{T}}(T)$ be T's guide mosaic drawing. Let $C(v)$ denote its configuration in $\mathcal{C}_{\mathcal{T}}(T)$, $C'(v)$ denote its guide configuration in $\mathcal{D}_{\mathcal{T}}(T)$ and c_t denote the square tiling position of a tile $t \in C(v) \cup C'(v)$. For each guide tile g find a unique tile $q \in C(v)$ such that c_q can be changed such that $c_q = c_g$, maximizing the number of g tiles that meet these relations while keeping $\mathcal{C}_{\mathcal{T}}(T)$ valid.

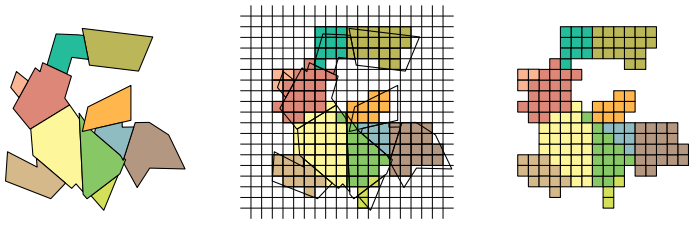


Figure 3.17: A rasterization of a scaled partition.

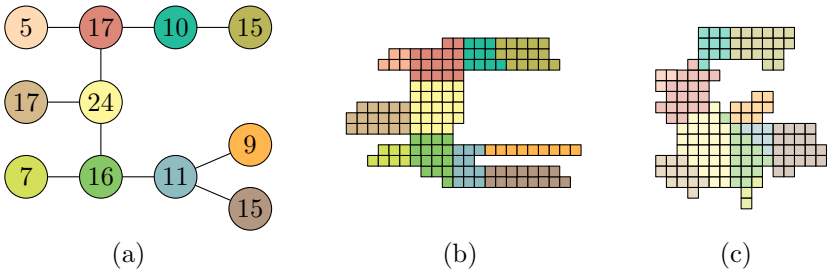


Figure 3.18: An arbitrary tree T (a), its mosaic cartogram composed of tiles (b) and its guide mosaic drawing composed of guide tiles (c).

For this maximization problem, we employ a heuristic consisting of a number of sequential procedures, on a per-node basis, maximizing the overlap of a node's guide tiles with the same nodes' tiles. The first procedure consists of moving a node's configuration at a time (bulk maximization) while other procedure consists of moving individual tiles of a node's configuration (fine maximization).

For this maximization problem, we employ a heuristic consisting of a number of sequential procedures, on a per-node basis, maximizing the overlap between a node's configuration and its guide configuration. The first procedure consists of moving a node's configuration at a time to maximize this overlap (bulk maximization), while the second consists of moving individual tiles of a node's configuration to maximize this overlap (fine maximization). The node order for which we apply this heuristic on is the same order described in Section 3.2.2 for creating a node's rasterization.

Once the heuristic has been applied for a specific node, we refer to this node as a *settled* node. Conversely, a node that is not yet settled is referred to as a *non-settled node*. Additionally, once a node becomes settled, it cannot become non-settled again, and its configuration is no longer allowed to be changed.

We define two basic translation operations used on a tile or set of tiles, which are needed for creating the discussed procedures.

- MOVE(S, d) - given a set of tiles S, they are translated once towards direction d on the square tiling, where d is one of the eight cardinal directions.
- PLACE(t, c) - given a tile t, it is taken from its original square tile position and is placed on the square tile position given by the coordinate tuple c.

Depending on their input, each one of these operation may break the mosaic cartogram properties. We are able to identify which mosaic cartogram property is broken using the methods described in Section 3.3.3. Depending on what which property is broken after performing one of the above operations, said operation is either undone, or the property is fixed by performing additional translation operations on non-settled nodes, which are also described in Section 3.3.3.

The heuristics described below are applied sequentially on the spinal nodes of T in breadth-first order, starting from the spine's head node. After all spinal nodes are settled, the same heuristics are sequentially applied on the non-spinal nodes in the breadth-first order of the rooted binary tree they belong to.

3.3.1 Bulk maximization. Maximizing initial guide tile overlap via pivoting

The first procedure involves maximizing guide tile overlap by translating the node's entire configuration at once. We do this by performing a sequence of MOVE$(C(v), d)$ for a given node v, where d is computed per each individual MOVE operation.

In order to do this, we use a settled node's configuration as a "pivot" in order to move a non-settled neighboring node's configuration around this pivot, until the non-settled node's tile and guide tile overlap is maximized. In order to avoid local maxima, we pivot a non-settled node's configuration in either a clockwise or counter-clockwise motion (depending on there the guide tile tiling positions), memorizing the amount of overlap after each sequential move, until the configuration is unable to move in the chosen motion due to breaking the mosaic cartogram's adjacency and hole rules (or until it returns to its original starting position), referred to as a *limit point*. When said limit point is reached, the non-settled node's configuration backtracks each sequential move applied to it until it reaches the place where its guide tile overlap was maximized and have its mosaic cartogram properties maintained. This procedure is illustrated in Figure 3.19.

While pivoting around a node, we explicitly allow for moves to introduce holes. This is because the configuration of a settled node is unpredictable, and pivoting around such a shape can temporarily introduce a hole in certain intermediary stages (Figure 3.20). If we forbid such movements, a node's configuration may not be able to reach its maximum overlap position, a position where if the node's configuration would reach, there may

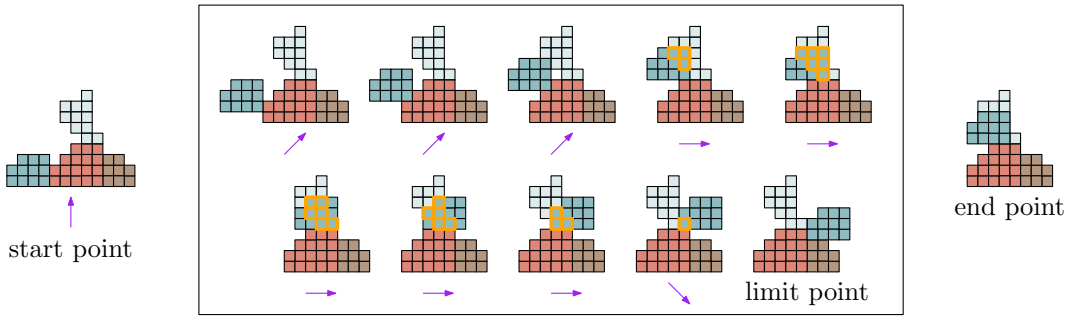


Figure 3.19: The pivoting procedure of a non-settled node (blue) around the settled pivot node (red) and its outcome. The light blue tiles are guide tiles, the orange highlighted tiles represent the overlap between a guide tile and a tile, and the purple arrow represents the next valid move direction.

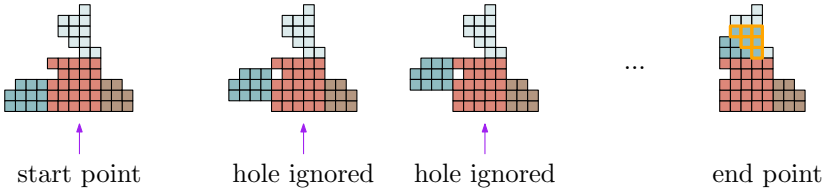


Figure 3.20: Pivoting the blue node around the red pivot node breaks while ignoring the hole rule of the mosaic cartogram, allowing for guide tile overlap.

be no holes that are introduced, maintaining the validity of the mosaic cartogram. If, however, holes are introduced in positions with maximum overlap, the configuration backtracks to the last known position where there was the most amount of overlap such that the resulting drawing is a mosaic cartogram.

We note that there is always a node to be used as a pivot, since the first node is the head node of a tree T's spine. This node already has some of its guide tiles overlapping with tiles by virtue of placing its guide configuration over its configuration centroid, described in Section 3.2.1.

However, as seen in Figure 3.21, if we only move the configuration of a non-settled node located in between another non-settled node and a settled node, an early adjacency rulebreak is inevitable, preventing this node's configuration to maximize its guide configuration overlap. In order to address this, we choose to move not only this non-settled node's configuration, but also the entire subtree rooted at this node, as seen in Figure 3.22. All nodes in this subtree are non-settled due to the breadth first order in which the heuristic is applied. This way, we are able to maintain the adjacency rule and obtain more guide configuration overlap than otherwise.

After maximizing the overlap for the node we applied this procedure on, we perform the fine maximization procedure in Section 3.3.2 on it. It is important to note that guide overlap is not guaranteed after applying this method.

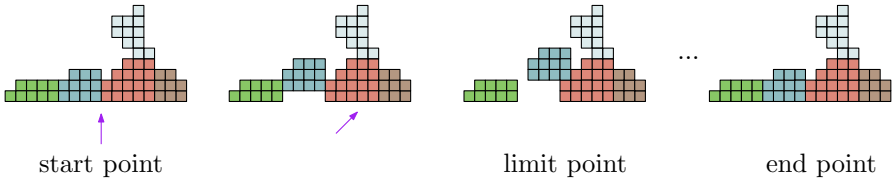


Figure 3.21: Attempting to move the blue node around the red pivot node breaks adjacency. The procedure reverts, and as a result it is unable to achieve any guide tile overlap.

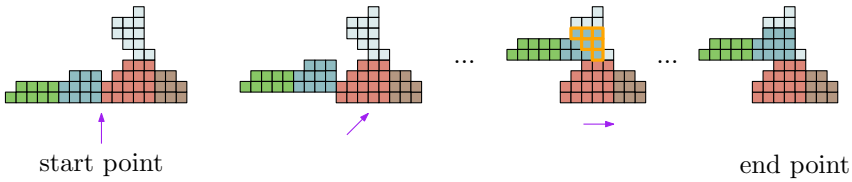


Figure 3.22: Moving the blue node along with its green child node around the red pivot node achieves guide tile overlap.

3.3.2 Fine maximization. Tile-by-tile shifting to maximize guide tile overlap

This procedure's goal is to further maximize a node's guide tile overlap on a per-tile basis, shaping our mosaic cartogram to match the guide mosaic drawing as much as possible. The goal of the previous procedure is to create as much guide tile overlap as possible, however any overlap at all is not guaranteed, in which case this procedure is not be able to create additional guide tile overlap.

Given the assumption that the given node's configuration has existing overlap with this node's guide configuration, we refer to guide tiles which do not overlap a tile as *candidate guide tiles*. Similarly, tiles that do not overlap a guide tile are referred to as *candidate tiles* (Figure 3.23). To promote mosaic cartogram shape cohesion, we prioritize incidence with a neighboring settled node's tiles. We achieve this by prioritizing guide tiles that are incident with the neighboring settled node's guide configuration. Additionally, we prioritize the candidate tiles that are the furthest from the configuration's centroid.

We now perform a PLACE(q, c_g) operations on each candidate tile q, for each candidate guide tile g, where c_g is g's square tiling position. When a candidate tile overlaps a candidate guide tile, they are removed from their respective candidate sets and this procedure is repeated until no more candidate guide tile can be overlapped, thus maximizing the number of overlapped guide tiles. We illustrate this in Figure 3.24.

After performing above steps, there are two relevant scenarios to consider. If the candidate tile set does not contain any more tiles, this means that all of them are now

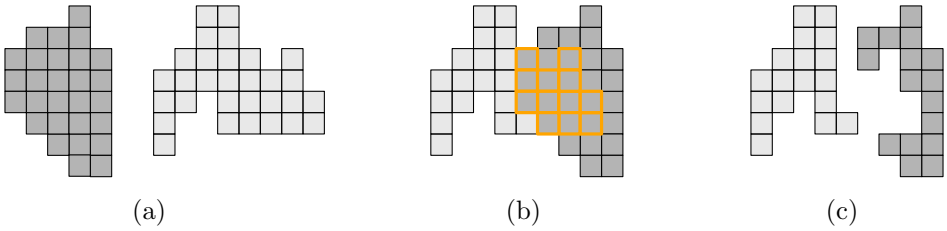


Figure 3.23: The configuration of a node (grey) along with its guide tiles (light grey) (a), an initial overlap between these tiles (highlighted in orange) (b), the remaining candidate tiles (grey) and candidate guide tiles (light grey) (C).

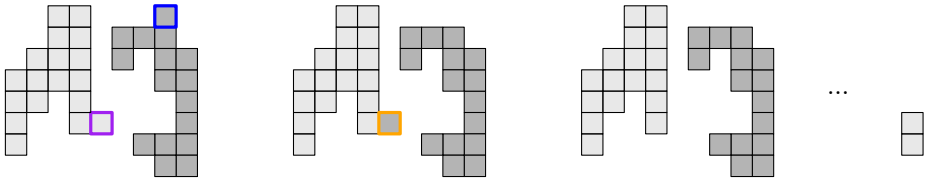


Figure 3.24: Given the scenario in Figure 3.23, we find a candidate tile (highlighted in blue) to overlap a candidate guide tile (highlighted in purple) (a), we overlap them (highlighted in orange) (b), and remove them form their respective candidate set (c), and repeat this procedure until the most guide tile overlap is achieved (d).

overlapping a guide tile, without breaking any mosaic cartogram rules. If the candidate tile set still contains tiles, this means that there are either no candidate guide tiles left, or the candidate guide tiles that are left would break the mosaic cartogram rules should they be overlapped by a tile. In the latter case, there is the possibility that the tile may overlap the guide tiles of non-settled nodes, lowering their maximum achievable overlap (as illustrated in Figure 3.25).

To account for this, the final step of this procedure is to identify the remaining candidate tiles which are overlapping the guide tiles of non-settled nodes, and performing a PLACE operation on each of them such that the described overlap is minimal and no mosaic cartogram rules are broken. Once this final step is performed (Figure 3.26), the node is considered to be a settled node. Tiles belonging to a settled node's configuration are prevented from being modified in any way by any translation operation invoked in the context of a non-settled node.

We note that the first node that this procedure is applied to is the head node of a tree T's spine. While this node is not affected by the previous bulk maximization procedure,

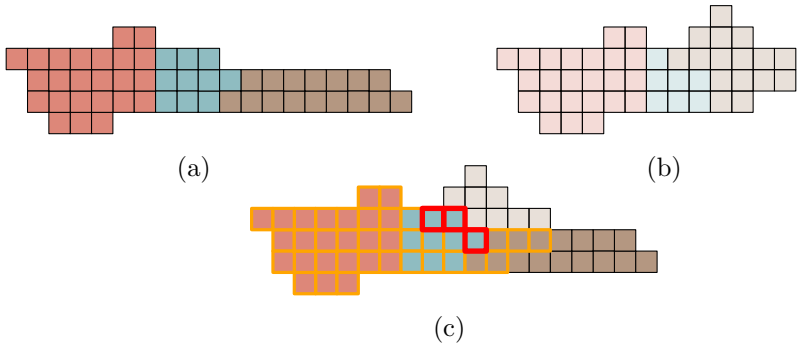


Figure 3.25: A mosaic cartogram (a) and its corresponding guide mosaic drawing (b), and their initial overlap (highlighted with orange) with three blue tiles overlapping three brown guide tiles (highlighted in red) (c).

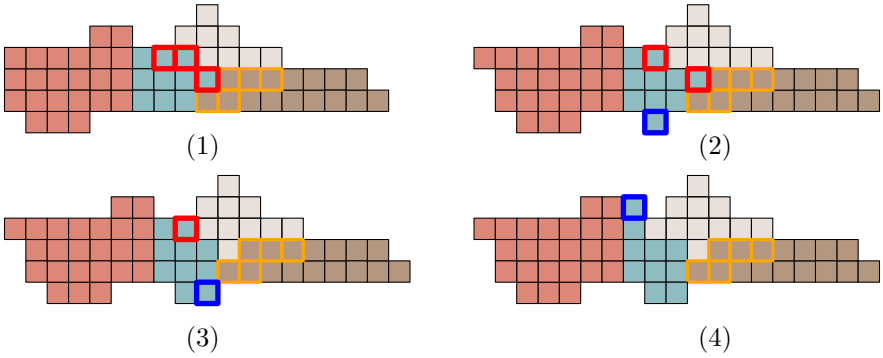


Figure 3.26: Moving each red-highlighted blue tile one by one in a valid position (highlighted in blue) that does not overlap any guide tiles and maintains mosaic cartogram rules.

it is not needed, since the prerequisite overlap already exists due to the placement of this node's guide mosaic drawing (following Section 3.2.1).

3.3.3 Mosaic cartogram rule breaking detection

In this section we discuss various methods used to check if the mosaic cartogram rules are broken after performing a translation operation. We list each rule that could be broken in which scenario and describe our methods of detecting it and correcting it through additional translations operations if possible.

Adjacency

Correct configuration adjacency is verified by checking for all nodes of T, if a tile t_v belonging to a node v is edge-connected to a tile t_u belonging to a node u, it should imply that v and u are adjacent in T. Additionally, if any tile t_v of v is not edge-connected to any tile t_u of u, it should imply that v and u are not adjacent in T. If both implications hold, the adjacency is correct. If not, adjacency is not correct. These implications are visualized in Figure 3.27. However, a distinction is made between these two implications.

If the first implication is not true, then the translation operation that introduced this rule break is undone. If the second implication is not true, we implement a way to correct it through performing additional translation operations on different tiles.

Considering the context of the second implication, given two nodes v and u that are adjacent in tree T, if a tile t_v of $C(v)$ with its initial square tile placement of c_v is placed in a different square tile placement c'_v such that by doing so, no tile of $C(v)$ are edge-connected with any tile of $C(u)$, both configurations $C(v)$ and $C(u)$ are now disconnected, breaking the adjacency rule. However, it is possible to correct this by performing a PLACE operation on a tile t_u such that t_u is placed at t_v's original square

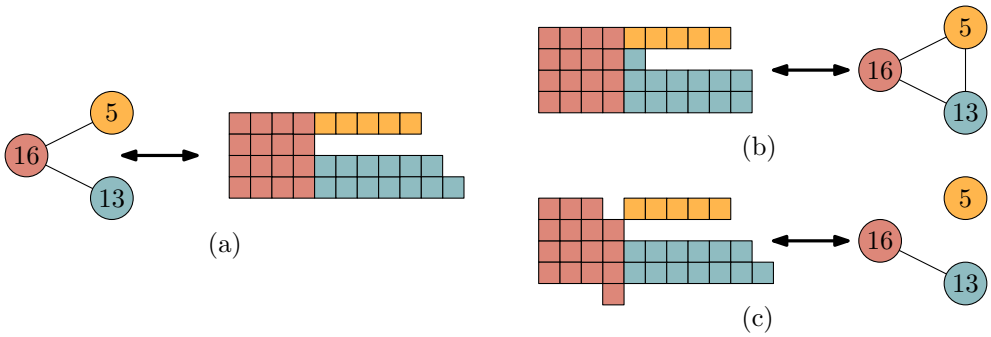


Figure 3.27: An arbitrary tree and a valid corresponding mosaic cartogram (a), a mosaic drawing where a blue node tile connects to a node it shouldn't connect with (b) and a mosaic drawing where the orange node's configuration is no longer adjacent with the configuration of the red node (c).

tile placement c_v without breaking any other mosaic cartogram rules. By doing so, the configurations $C(v)$ and $C(u)$ are adjacent again, correcting the adjacency rule. If no such tile t_u can be found, then the translation operation that introduced this rule break is undone. A successful application of this correction can be seen in Figure 3.28.

Contiguity

To check if a given node v's configuration $C(v)$ is contiguous, we perform a depth first search starting from an arbitrary tile t in $C(v)$ and recursively visiting each tile's edge connected neighbor tiles, counting each visited tile. If the number of visited tiles is equal to the number of tiles in $C(v)$, then $C(v)$ is contiguous. Otherwise, $C(v)$ is not contiguous and the translation operation is undone. This is illustrated in Figure 3.29.

Overlap

Checking for overlap is a relatively simple operation. Given two tiles t_1 and t_2, if their square tiling positions c_{t_1} and c_{t_2} are equal, this means that an overlap is present.

This rule break is verified both in the bulk maximization procedure as well as the fine maximization procedure. If a translation operation is done that induces unintended

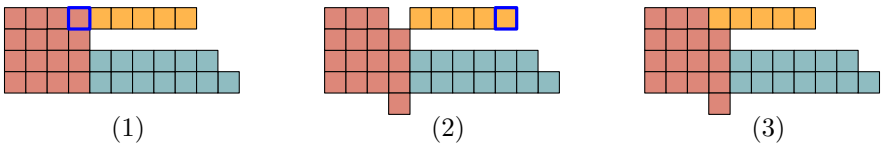


Figure 3.28: Given the mosaic cartogram in Figure 3.27, by moving the red tile highlighted in blue we cause an adjacency violation, which we fix by moving an orange tile (highlighted in blue) to where the first red tile was moved away from.

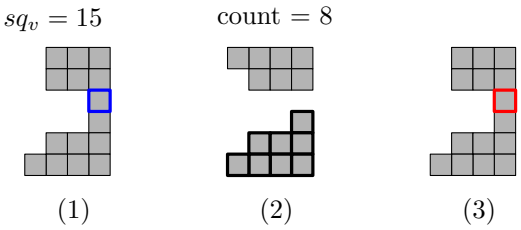


Figure 3.29: Moving away the tile highlighted in blue causes the configuration to be discontiguous, since the count is not equal to the number of tiles before the move (tiles highlighted in black), thus the move is undone (highlighted in red).

overlap, then this translation operation is undone.

There are certain cases where an overlap is intended, but only temporarily. Given a node v that is currently being settled, we may want to translate a tile t_v of this node's configuration via the PLACE operation such that it overlaps a guide tile of the same node, however, t_v would overlap with another tile t_u that belongs to the configuration of a non-settled node u. If node u has correct adjacency with v, we want to move t_u away such that the overlap is eliminated.

We do this by now performing an operation PLACE on t_u, such that it overlaps a tile t_w, that belongs to the configuration of a non-settled node w and maintains correct adjacencies after performing it. We then repeat this for t_w recursively until there is no other tile belonging to a node to be overlapped, in which case this tile would be placed on a position where no other rules are broken. If no such position exists, the chain of PLACE operations are recursively undone. A successful application of this procedure is illustrated in Figure 3.30.

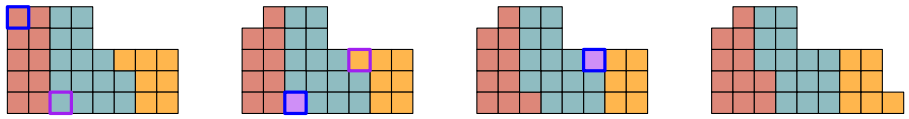


Figure 3.30: We highlight in blue the tiles that we want to move, and we highlight in purple the tile that will be overlapped. An overlap between two tiles is shown as a purple tile. In this sequence, we first move a red tile to overlap a blue tile, then move the overlapped blue tile to overlap an orange tile, and lastly move the overlapped orange tile such that no mosaic cartogram rules are broken.

Holes

The detection of holes is checked only for the PLACE operation. For this algorithm, we assume that the mosaic cartogram is contiguous. The way we check for holes is as follows:

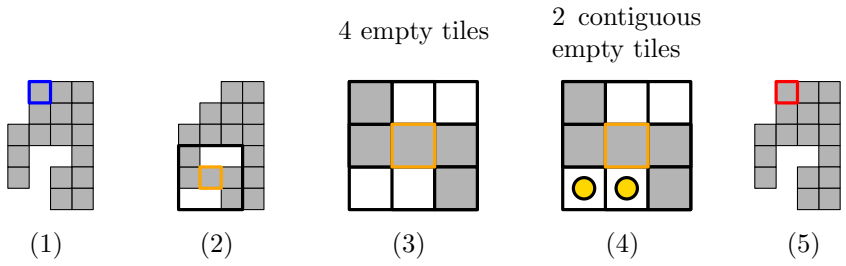


Figure 3.31: We move the blue highlighted tile in a place where it would create guide tile overlap (highlighted in orange), however it would introduce a hole, as there are only 2 contiguous empty tiles instead of 4, if no hole were to have been introduced, and thus we undo the move (highlighted in red).

After placing a tile t at a square tiling placement c_t, we inspect a 3 by 3 area of tiles around c_t, choose an arbitrary square tiling placement c_r that does not contain a tile and recursively count the number of edge-connecting square tiling placements neighboring r that also do not contain a tile. If this count is the same as the number of square tiling placements that do not contain a tile withing this 3 by 3 area, then a hole has not been introduced by placing tile t. Otherwise, the placement of t is undone. This procedure is illustrated in Figure 3.31.

A hole can be intentionally introduced in a certain situation, but only temporarily, when performing a PLACE operation on a tile. If a tile t_v of $C(v)$ with its initial square tile placement of c_v and four edge-connected neighboring tiles, is placed in a different square tile placement c'_v, the empty square tiling placement left at c_v is a hole. This hole can be filled using another PLACE operation of a tile t' that does not have four edge-connected neighbors such that no other mosaic cartogram rules are broken. If no such tile t' exists, the translation operation of t_v is undone.

When moving an entire set via a MOVE operation, the same method discussed above does not work. While computationally expensive, depending on how space-efficient the mosaic cartogram is, it is possible to check for a hole using a depth first search algorithm on an arbitrary empty square tiling, counting the number of edge-connected empty square tilings and comparing the result with the total number of empty square tiles in a given bounding box around the mosaic cartogram.

Chapter 4

Conclusions

In this chapter, we apply our methodology on a data set which Meulemans et al. [1] also used, namely the municipality map of the Netherlands. We note that our method is only applied for mainland Netherlands. We investigate the following aspects: parametrization of the partitioning step and spine selection, presentation of relevant intermediary results, a comparison to state-of-the-art, the limitations of our method, ending with a discussion and future work.

4.1 Parametrization and intermediary results

We specify the parameters that have been used in the case of our dataset: a safe productivity value of 15 and a dilation ratio of 5. This produces the partition seen in Figure 4.1. In the same figure, we illustrate our choice of what nodes the spine should be composed of.

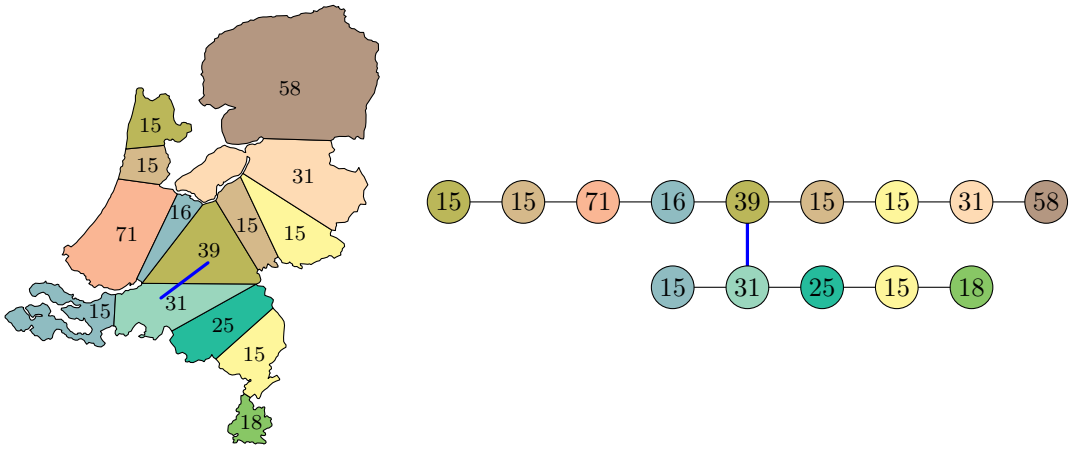


Figure 4.1: The resulting partition of mainland Netherlands as well as its corresponding tree with the spine highlighted in blue.

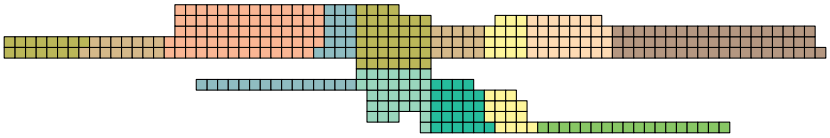


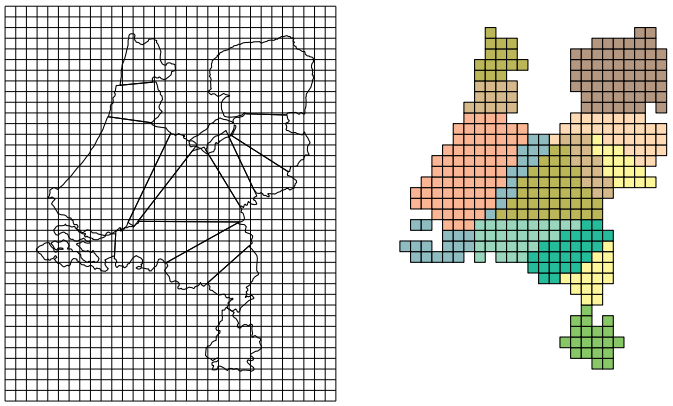
Figure 4.2: The resulting mosaic cartogram to be used as a starting point.

Given this parametrization, we showcase key relevant intermediary outputs that are related to the methods described in this thesis. Figure 4.2 shows the mosaic cartogram used as a starting point for shaping. Figure 4.3 shows the scaled regions of the partition obtained in Figure 4.1, as well as its corresponding rasterization, used as a guide to shape our mosaic cartogram.

4.2 Comparison

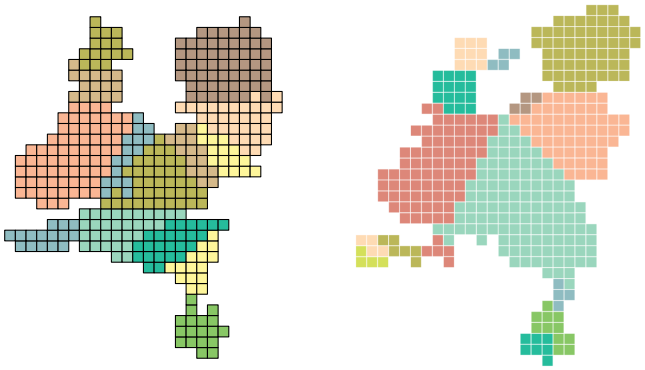
The most relevant state-of-the-art solution to this problem consists of the solution proposed by Meulemans et al. [1]. Our final result is seen in Figure 4.4, as well as the chosen result of Meulemans et al.

We can see that the quality of our result is roughly comparable with the state-of-the-art solution. We have added the guarantee that a mosaic cartogram is always generated for no significant shape trade-off in the case of this dataset, however the shape is not guaranteed to be optimal in all cases. We discuss potential ways to improve our algorithm in Section 4.4.



(a) (b)

Figure 4.3: The scaling and placement of the partition (a) and its rasterization (b).



(a) (b)

Figure 4.4: Our shaped mosaic cartogram (a) side by side with the mosaic cartogram obtained by Meulemans et al. [1].

4.3 Limitations and shortcomings

One of the limitations of our methodology is that the process for creating a result is not fully automated. While this is also the case for Meulemans et al. [1] in regards to the parametrization step, our method assumes the parametrization of the spine as well. Additionally, due to the lack of further experimentation, various steps of our heuristic can fail due to unforeseen scenarios that are not accounted for since they were not encountered in our application of the algorithm. Another limitation of our methodology is that the shape is not always optimal, however a valid mosaic cartogram is always produced.

4.4 Discussion and Future Work

Our work consist of improving the existing pipeline that Meulemans et al. [1] proposed. The advantage of their pipelined approach is observable within our results and the same advantage is still maintained. While one of the concerns of their work was that the resulting grid map would have a jagged boundary, it still remains a valid concern in our pipeline as well, since there is no specific method that addresses it. Thus, an avenue of improvement of the grid map quality remains to be exploited in the form of adding an extra post-processing step after our result is complete.

We also find an avenue of improvement regarding the creation of guide tiles that are used to shape the mosaic cartogram. Further improvement can be made by creating a higher quality rasterization of the partition by incrementally scaling a specific region until the exact number of guide tiles are generated. Taking this process a step further, the mosaic cartogram rule breaking detection methods discussed in Section 3.3.3 could be applied on this mosaic drawing to potentially convert it into a mosaic cartogram. An example of such a correction would be to remove certain guide tiles from our mosaic

drawing that break adjacency rules (since we know the correct adjancencies from the dual graph of the partition) and adding new guide tiles that don't break this rule to replace them. While this method does not guarantee a mosaic cartogram, it can use our mosaic cartogram generation and shaping methods as a fallback.

Additional experimentation with this pipeline can also help to determine more optimal results based on experimentation with the parametrization step, including figuring out an algorithm that is able to find a well suited spine for a partition. However, the core problem remains that it is difficult to quantify the quality of a grid map and automatically determining the input map's salient features.

A node-by-node shaping procedure of the mainland Netherlands data set can be found in Appendix A.

References

[1] W. Meulemans, M. Sondag, and B. Speckmann. "A Simple Pipeline for Coherent Grid Maps". In: *IEEE Transactions on Visualization and Computer Graphics* 27.2 (Feb. 2021), pp. 1236–1246. DOI: 10.1109/tvcg.2020.3028953. URL: https://doi.org/10.1109/tvcg.2020.3028953.

[2] R. G. Cano, K. Buchin, T. Castermans, A. Pieterse, W. Sonke, and B. Speckmann. "Mosaic Drawings and Cartograms". In: *Computer Graphics Forum* 34.3 (June 2015), pp. 361–370. DOI: 10.1111/cgf.12648. URL: https://doi.org/10.1111/cgf.12648.

Appendix A

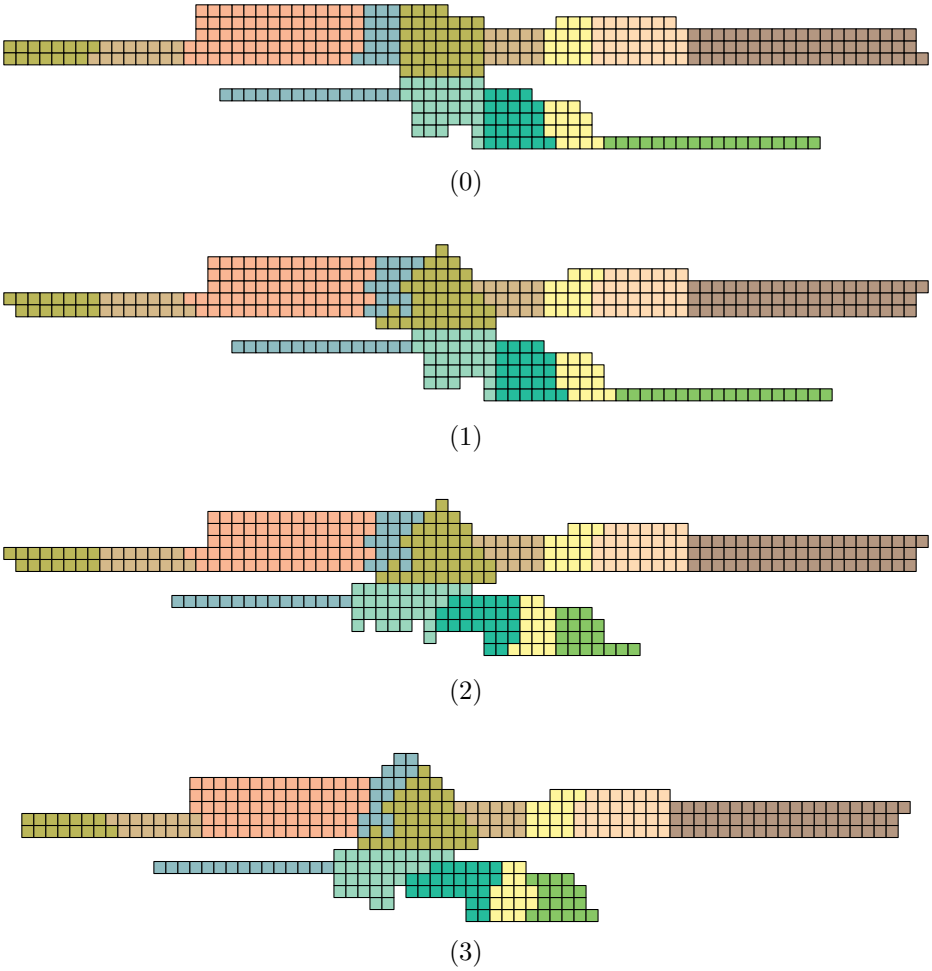
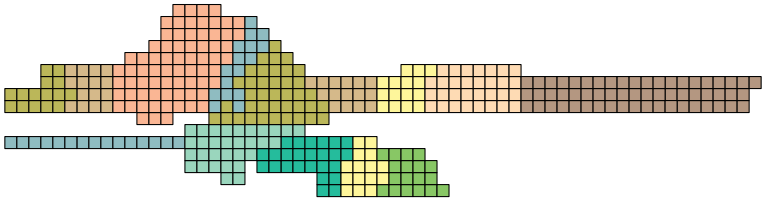
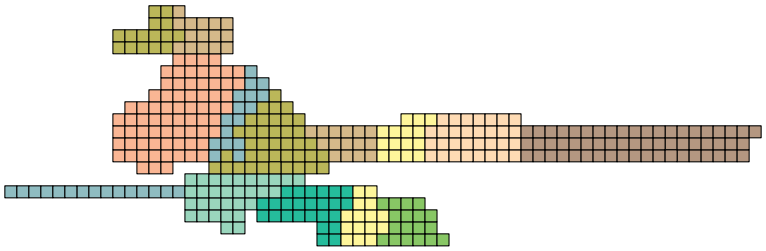


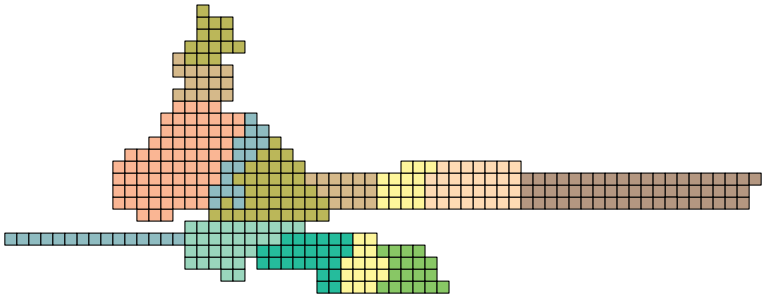
Figure A.1: Unshaped mosaic cartogram (0) and the heuristic algorithm output for the first 3 non-settled nodes (1-3).



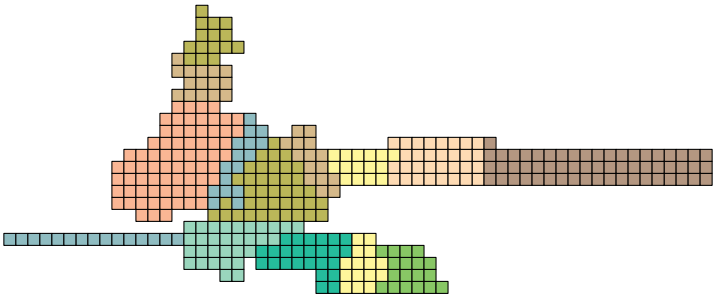
(4)



(5)

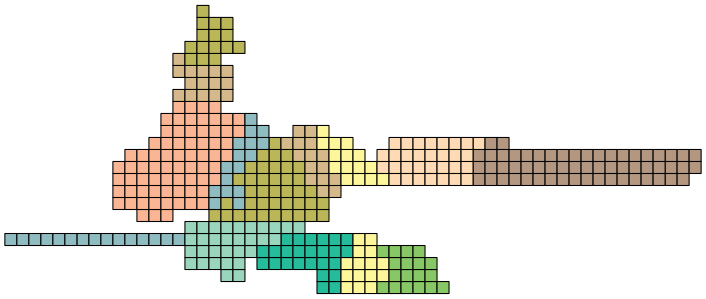


(6)

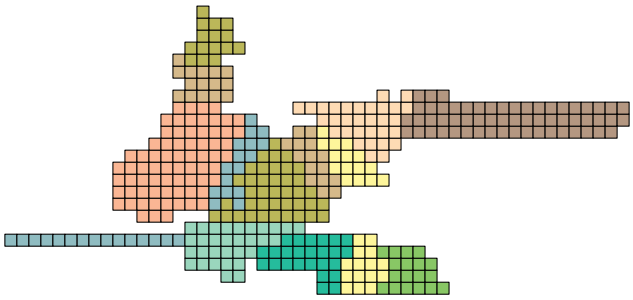


(7)

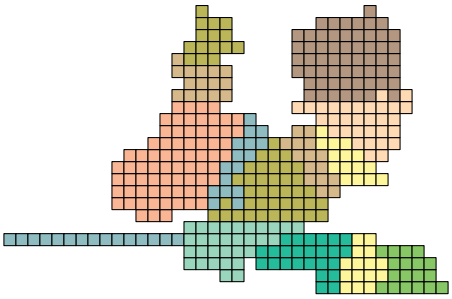
Figure A.2: Heuristic algorithm output for the next 4 non-settled nodes (4-7).



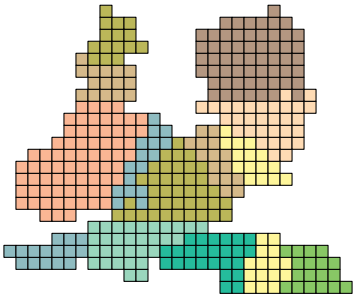
(8)



(9)

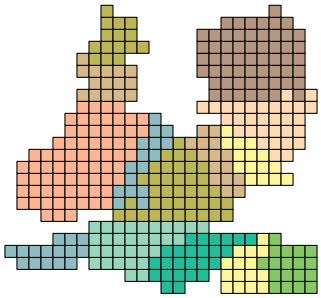


(10)

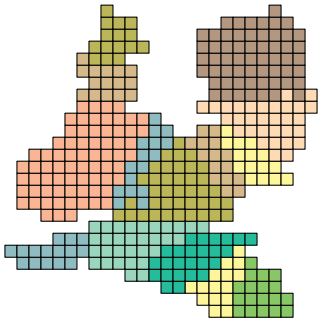


(11)

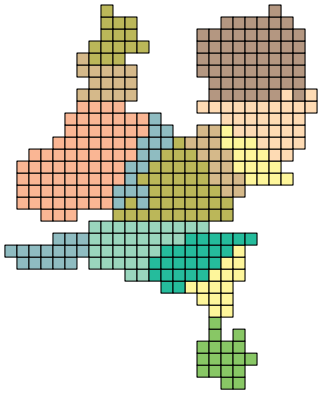
Figure A.3: Heuristic algorithm output for the next 4 non-settled nodes (8-11).



(12)



(13)



(14)

Figure A.4: Heuristic algorithm output for the last 3 non-settled nodes (12-14).