Equal-area Breaks: A Classification Scheme for Data to Obtain an Evenly-colored Choropleth Map

Anis Abboud Hanan Samet Marco D. Adelfio

Department of Computer Science, University of Maryland
College Park, MD 20742, USA
{anis, hjs, marco}@cs.umd.edu

ABSTRACT

A classification scheme for choropleth maps for a spatially varying property called *equal area* is introduced that aims to obtain a coloring for the map such that the screen area associated with each color is equal. As some regions are larger than others (e.g., Russia vs. Switzerland in a choropleth country map), naively assigning an equal number of regions with a particular property value to each color could result in the map being dominated by one or a few colors, and with the possibility that the other colors barely discernible. The goal is to assign the ranges of the property values to colors so that the total area of the regions associated with each color is roughly equal thereby rendering a more symmetric and appealing visualization. A number of algorithms to achieve an equal-area assignment to regions are presented and evaluated from both a visual, via the aid of a user study, and a computational complexity perspective. They include greedy algorithms, as well as an optimal algorithm based on dynamic programming, and are compared with other approaches for assigning colors to regions, such as the Jenks natural breaks optimization algorithm. The final algorithm is a modified approach which tries to simultaneously balance the goal of equal area for each color with that of assigning an equal number of regions to each color with the result that works well for both properties corresponding to absolute data and area-normalized data such as densities.

Categories and Subject Descriptors

H.5.2 [**Information Interfaces and Presentation (e.g., HCI)**]: User Interfaces—*map visualization*; I.1.2 [**Symbolic and Algebraic Manipulation**]: Algorithms—*sequence partitioning*

General Terms

Algorithms, Maps

Keywords

Data classification, natural breaks, map visualization, sequence partitioning

1. INTRODUCTION

Maps are used primarily for navigation and to visualize how measurements (whether they are nominals, ordinals, ranges, or ratios) vary across a geographic area. In this paper we are concerned with the visualization of spatially-varying ratio measurements (i.e., numbers such as population, population density, per capita income, etc.). Such maps are known as *choropleth* maps [1] (e.g., [17]). The region boundaries are predefined although they can vary in size and scope (e.g., city, county, country, continent, world, etc.). This is in contrast with regions whose boundaries are defined by the data values.

Choropleth maps are used to represent two types of data: spatially extensive and spatially intensive [1]. Spatially extensive data are cumulative data such as population. On the other hand, spatially intensive data are concepts like rates, densities, and proportions which reflect the application of a normalization process over an area such as an average.

One of the keys to understanding a choropleth map is the classification process/method used to differentiate between the data values. The data values can be broken up into classes depending on the ranges of the data values so that they correspond to intervals that are equal in magnitude (e.g., five classes of one million each for data ranging between 0 and five million), equal in cardinality or length (e.g., five classes of 40 each for data corresponding to a total of 200 countries) also known as quantiles, as well as equal in area (e.g., five classes of 10,000 square miles each for data ranging between 0 and 50,000 square miles).

The equal-area classification method, which represents our contribution, is designed to overcome the general drawback of the other classification processes, which, by not taking the area into account, can result in the map being dominated by one or a few colors, with the other colors being barely visible. For example, consider Table 1 which contains the countries/territories in South America, and their population as of July 2009 [7] We use three colors - yellow, orange, and red, where the darker color indicates a larger population. One way to assign the colors is to use yellow for the first five regions, orange for the next five, and red for the last five, and shown in Figure 1a, which correspond to quantiles. The problem is that the five countries colored yellow have a much smaller area than the five countries colored red, and therefore the map does not look very evenly colored.

The equal-area method assigns the colors so that the screen areas spanned by each color are roughly the same and shown in Figure 1b. This is achieved by computing the areas of all regions and finding the "breaks" (where to split between colors) that achieve this coloring. In this paper we show that this can be achieved by a number of algorithms of varying complexity and visual effect. Of course, an alternative approach to avoid the uniform coloring is to use area-normalized data values such as a density (recall the distinction made above between spatially extensive and spatially intensive data).

	Country/Territory	**Population**
1.	South Georgia and the South Sandwich Islands (UK)	20
2.	Falkland Islands (UK)	3140
3.	French Guiana (France)	221500
4.	Suriname	472000
5.	Guyana	772298
6.	Uruguay	3477780
7.	Paraguay	6831306
8.	Bolivia	9863000
9.	Ecuador	14573101
10.	Chile	16928873
11.	Peru	29132013
12.	Venezuela	31648930
13.	Argentina	40482000
14.	Colombia	45928970
15.	Brazil	191241714

Table 1: Population of countries/territories in South America as of July 2009

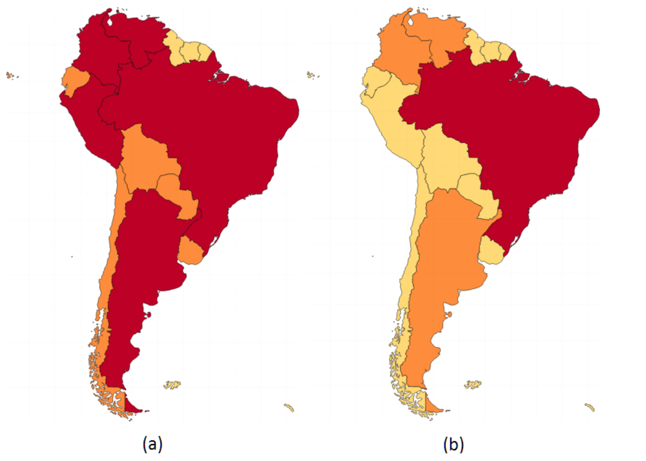


Figure 1: Visualizing the data in Table 1 on the map with equal-length/quantiles (a), and equal-area (b).

The simple classification methods that we mentioned (equal-intervals/equal-steps and equal-length/quantiles) have major drawbacks. In particular, in the equal-intervals process, while being the only deterministic method, the intervals that are used can be completely unrelated to the actual data values possibly resulting in having very few, if any, members in the class. The equal-length method attempts to overcome this by making sure that there is an equal number of regions in each class. However, as we mentioned earlier, by ignoring the nature of the data, this simple approach is not always adequate.

In our discussion of the equal-area method we pointed out the need for considering all the possible "breaks" combinations which is a problem of exponential complexity. Therefore, we have to devise smart algorithms to find the breaks in which we are interested. Besides the above classification methods there is a range of other data classification methods for statistical mapping, such as geometric progressions, standard deviation and Jenks natural breaks [10, 12, 14], that are used for finding "breaks". The Jenks natural breaks method is used to classify data values into different classes according to the breaks or gaps that naturally exist in the data. Another alternative is the head/tail break method [15] which is designed to deal with distributions that have a heavy tail such as power laws. In other words, it recognizes the fact that there are far more objects of small magnitude than objects of large magnitude. The Jenks and head/tail breaks methods focus on clustering the values associated with the regions into similarly-valued chunks, which are visualized with the same color.

In this paper, we focus on clustering the areas corresponding to the regions into chunks with roughly equal total area, aiming to achieve an evenly-colored choropleth map. We derive a number of algorithms for its computation paying close heed to their complexity. We show that the equal-area method serves as a reasonable compromise between the drawbacks of the other methods both visually and via the aid of a user study. In particular, our results for both absolute raw data values (spatially-extensive data) and density data (spatially-intensive data) show that it works quite well for both.

We note that some variant on the equal-area data classification was available in the Esri ArcView 3.x commercial GIS software, but for unknown reasons it was later dropped in the successor software ArcMap. No public details are known about the algorithm used in ArcView 3.x.

It is interesting to note that the equal-area classification process is related to the Cartogram method [19] in the sense that the equal-area method (as well as the other methods that we present) reflect the relative values of the data by varying the colors of the regions, while the Cartogram method does this by varying their displayed area while preserving their shape, hence they do not need the aid of color.

The contributions of our paper are:

1. Equal-area classification method for coloring a choropleth map that overcomes the drawbacks of the equal-length method and the various algorithms for its computation including an optimal one that makes use of dynamic programming.

2. A new classification method for coloring a choropleth map that enables users to combine the benefits of the equal-area and equal-length methods using a parameter analogous to the f-score from information retrieval that combines precision and recall [18]. In our problem domain this parameter, termed a *W-score*, enables the method to yield results that are in sync with those of a user study for both spatially extensive and spatially intensive data.

The rest of this paper is organized as follows. Section 2 describes a number of algorithms to compute the equal-area classification method. Section 3 contains the result of an evaluation of the algorithms from a visual perspective as well as the results of a user study. Section 4 uses the results of the user study to devise an optimized algorithm that is a blend between the equal-length and equal-area methods. Section 5 contains concluding remarks and directions for future work. It is worthy to note that this paper was motivated by our development of a tool that enables users to create map visualizations out of a data tables that they provide [8]. In particular, one component of this system automatically generates a legend and assigns colors to regions of choropleth maps.

The paper contains a number of maps all of which were rendered using the D3 JavaScript library [3]. In addition, we used its capability to compute a region's area in order to find the area in pixels that each country occupies on the screen (as opposed to the actual area in km^2 on the spherical surface of the Earth) because the areas change under the various projections. We use the Winkel-Tripel [9] projection in world our maps, as it is regarded as one of the best world projections [11], but it's important to note that the projection used makes no difference in what we present, and Mercator projection would have fulfilled the same purpose in conveying the ideas.

Our examples make use of color progression to depict the data. There are many possible types available (e.g., [17]). These include single hue, bipolar hue, complimentary hue, blended hue, partial spectral, full spectral, and value (for monochromatic which means white to black). We use the blended hue color progression which blends two endpoint hues (yellow and brown in our case) to obtain related hues. The colors that we used were selected using the ColorBrewer [2] system as shown in Figure 2.

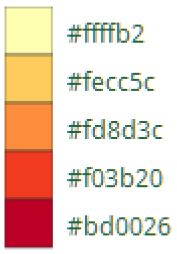


Figure 2: Blended hue progression for the yellow-brown color pair used in our maps - from ColorBrewer [2].

2. ALGORITHMS

2.1 Problem Definition

Given a sequence of N numbers (the areas of the regions, sorted by the values corresponding to the property associated with the regions–e.g., population), and a number K (of ranges/colors), the goal is to partition the sequence into K parts, so that the sum of the areas of each part is roughly equal.

2.2 Simple Example

Given the numbers [11, 30, 84, 146, 251, 214, 256, 476, 1255, 930, 1192, 1149, 1338, 4307, 8994] (the areas in pixels on the screen of the N=15 countries/territories in South America from the example in the introduction), and K=3 colors (yellow, orange, and red), we aim to partition the sequence into 3 chunks, with sums that are as close as possible to each other. The sum of all 15 numbers above is 20634, and therefore the sum of an average chunk is 20634/3 = 6878. Our goal is to partition the sequence into chunks whose sum is as close as possible to this average, as can be seen in Figure 3.

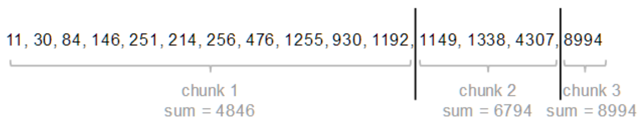


Figure 3: Simple chunks example.

2.3 Greedy Algorithms

The simplest algorithm that comes to mind is to build the chunks one-by-one while traversing the sequence of numbers from left to right. We start inserting numbers into the chunk until the sum of the elements in the chunk exceeds the target/average chunk sum, at which time we close it and start a new chunk.

2.3.1 Pseudocode

Algorithm 1 Greedy algorithm for partitioning a sequence of N numbers into K chunks with roughly an equal sum. Complexity: $O(N)$.

```
 1: function GreedySplit(numbers, K)
 2:     average_chunk ← $\frac{\text{sum}(numbers)}{K}$
 3:     chunks ← <empty list of chunks>
 4:     new_chunk ← <empty list of numbers>
 5:     for number in numbers do
 6:         Append number to new_chunk.
 7:         if sum(new_chunk) $\geq$ average_chunk then
 8:             Append new_chunk to chunks.
 9:             new_chunk ← <empty list of numbers>
10:     Append new_chunk to chunks.        ▷ The last chunk.
11:     return chunks
```

2.3.2 Observation

Since the above greedy algorithm always overestimates the chunks (i.e., keeps inserting in the chunks until the sum of the elements in the chunk exceeds the average chunk sum), the last chunk (the "leftovers") will be significantly smaller than the others. To mitigate this issue, we can consider the total sum so far in order to decide when to start a new chunk.

2.3.3 Modified Pseudocode

Algorithm 2 Modified greedy algorithm for partitioning a sequence of N numbers into K chunks with roughly equal sum. Complexity: $O(N)$.

```
 1: function GreedySplit2(numbers, K)
 2:     average_chunk ← $\frac{\text{sum}(numbers)}{K}$
 3:     chunks ← <empty list of chunks>
 4:     new_chunk ← <empty list of numbers>
```

```
5:   $num\_chunks \leftarrow 1$
6:   for $number$ in $numbers$ do
7:     Append $number$ to $new\_chunk$.
8:     if sum(numbers up to number) $\geq$
9:         $average\_chunk \times num\_chunks$ then
10:      Append $new\_chunk$ to $chunks$.
11:      $new\_chunk \leftarrow$ <empty list of numbers>
12:      $num\_chunks{+}{+}$
13:  Append $new\_chunk$ to $chunks$.        ▷ The last chunk.
14:  return $chunks$
```

2.4 Optimal Dynamic Programming Algorithm

Before we present an optimal algorithm, let us define the criterion which we are using to measure the performance of an algorithm that returns a set of K chunks that partition the sequence of numbers:

$$ERROR\,(breaks) = \frac{1}{K} \cdot \sum_{i=1}^{K} |sum\,(chunk_i) - AVG|$$

where $AVG = \frac{sum\,(numbers)}{K}$ is the sum of the average/optimal chunk, and $sum(chunk_i)$ is the sum of the numbers in the i^{th} chunk of the partitioning.
In other words, the formula computes the average distance between the sum of a chunk in the given partitioning, and the optimal chunk sum. An optimal algorithm would find a set of breaks to partition the list with the minimal $ERROR$.

2.4.1 Key Idea 1

The first key idea in the algorithm, is that we can find an optimal position for the last break with a linear pass on the numbers. We can start traversing the list from right to left and summing up the numbers, until they add up to at least AVG (let the last number we include be x). Since the numbers are arbitrary, it's unlikely that they will add up to AVG exactly, so the sum will be slightly greater than AVG if we include x in the last chunk and slightly less than AVG if we do not include x in the last chunk.

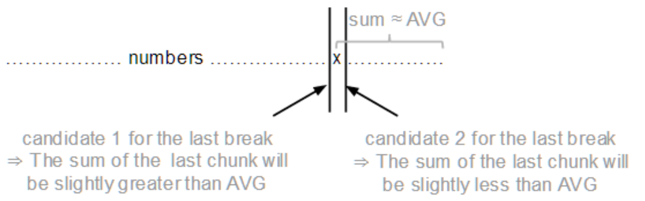

sum ≈ AVG
numbers
x
candidate 1 for the last break
⇒ The sum of the last chunk will be slightly greater than AVG
candidate 2 for the last break
⇒ The sum of the last chunk will be slightly less than AVG

Figure 4: Two candidates for an optimal position of the last break

Observation There is an optimal partitioning in which the last break is around x – either before it or after it.

Proof
Given an optimal partitioning of the list, if the last break is around x, then we are done. Otherwise, there are two options for the last break:

1. The last break is before candidate 1. In this case, we argue that moving it to where candidate 1 is can only improve the error. Notice that moving the last break to the right modifies only the last two chunks, call them A and B. The sum of chunk A will increase by some δ (the sum of the numbers between the current break and candidate 1), and the sum of chunk B will decrease by the same δ. Since the sum of chunk A was greater than AVG and is still greater, but now is δ closer to AVG, the error stemming from it will decrease by exactly $\frac{1}{K} \cdot \delta$. Since the sum of chunk B changed by δ, the error stemming from it can increase by at most $\frac{1}{K} \cdot \delta$. Therefore, the error in the new partitioning can only decrease. Hence, the new partitioning is also optimal.

2. The last break is after candidate 2. In this case, we argue that moving it to where candidate 2 is can only reduce the error. Similarly to the other case, the error from the last chunk will decrease by exactly $\frac{1}{K} \cdot \delta$. The only caveat here is that in the given optimal partitioning, there might be multiple breaks after candidate 2. In this case, we will move them one by one to where candidate 2 is, starting from the leftmost one. Every time we shift a $break_i$ to the left, the error from the chunk on its right will decrease by some $\frac{1}{K} \cdot \delta_i$, and the error from the chunk on its left can increase by at most $\frac{1}{K} \cdot \delta_i$. Therefore, the total error cannot grow. Hence, the new partitioning is also optimal.

■

The pseudocode for a procedure to find the two candidates described above is given below.

```
1: function FindLastBreakCandidates($numbers$)
2:   $s \leftarrow 0$                    ▷ Sum of the numbers so far.
3:   for $i = N - 1$ until 0 do
4:     $s \leftarrow s + numbers[i]$
5:     if $s > AVG$ then
6:       return $(i,\ i + 1)$
```

2.4.2 Key Idea 2

Let $BestError(m, b)$ be the minimum error we can get for placing b breaks to partition the first m elements of the list (which has N total numbers), and $Breaks(m, b)$ be the set of corresponding breaks. We are interested in $Breaks(N, K - 1)$.
Building on top of Key Idea 1, we can define the following recursive relation/function:
$BestError\,(m, p) =$

1. Find the two candidate locations for the last pivot, call them loc_1, loc_2.

2. Compare

   - $BestError(loc_1, p - 1)$ $+|sum(\text{numbers between } loc_1 \text{ and } m) - AVG|$
   - $BestError(loc_2, p - 1)$ $+|sum(\text{numbers between } loc_2 \text{ and } m) - AVG|$

   and choose the break with the smaller error.

In other words, for each of the two candidates for the location of the last break, we recursively find the other breaks, and then we compare the error from the two options and choose the better candidate. Calculating this recursively will have a high computation cost, as we might repeat many computations. Therefore, we will compute it using dynamic programming. In particular, we allocate a two dimensional array to store the values of $Breaks(m, b)$ for all $m \in [0, N], b \in [0, K-1]$, and compute them column-by-column so that whenever we need a value for $Breaks(loc_{1 \text{ or } 2}, b-1)$ we can look it up in the table in constant time.

2.4.3 Key Idea 3

As described in Key Idea 1, finding the candidates for the last break takes linear time, as we need to traverse the list from right to left. However, as we are using dynamic programming to compute all the values for $Breaks(m, b)$ (for all values of m and b), we can save on some computations. Notice that if the candidates for the rightmost break in $Breaks(m, b)$ were around position x, then the candidates for the rightmost break in $Breaks(m-1, b)$ have to be on the left of x (they can't be on the right), because the list is shrinking from the right, so the rightmost chunk needs to grow from the left to remain close to the average. Thus the break needs to move left. Therefore, we can compute the $Breaks(m, b)$ in a loop where we move m and the candidates backward simultaneously... until they hit index 0. This way we compute N cells in the $Breaks(m, p)$ array in $O(N)$ time.

2.4.4 Key Idea 4

In the ideas above, we use the "sum" function, which if implemented naively, would take linear time to compute. However, by preprocessing the sequence of numbers and building the "cumulative sum" array, known as the "prefix sum" (i.e., given the list of numbers: $[a, b, c, d, e]$, we build: $[0, a, a+b, a+b+c, a+b+c+d, a+b+c+d+e]$), we can build a function that returns the sum of any subsequence between indices i and j in constant time:

Algorithm 3 Prefix sum array calculation. PREPROCESS(*numbers*) is invoked once on the original array, to compute the prefix sum array in $O(N)$ time. After that, PSUM(i, j) can be invoked to compute the sum of the numbers between any two indices i (inclusive) and j (exclusive) in the original array in $O(1)$ time.

```
1: function Preprocess(numbers)
2:     $sums\_array \leftarrow [0]$
3:     s ← 0
4:     for $x$ in $numbers$ do
5:         $s \leftarrow s + x$
6:         Append $s$ to $sums\_array$.
7:     return $sums\_array$

1: function PSum($i$, $j$)
2:     return $sums\_array[j] - sums\_array[i]$
```

It is important to note that from now on, when we invoke PSUM(i, j) in the pseudocodes, we assume that the preprocessing above has been performed, and we call PSUM(i, j) to calculate the sum of the numbers between *indices* i (inclusive) and j (exclusive) in constant time. Indices are 0-based.

2.4.5 Pseudocode for the Dynamic Programming Optimal Algorithm

Using the above ideas, Algorithm 4 is the pseudocode for the optimal algorithm to find breaks that partition a sequence of N numbers into K chunks with a roughly equal sum:

Algorithm 4 Dynamic Programming optimal algorithm for finding the breaks (i.e., the indices at which to split the sequence) for partitioning a sequence of N numbers into K chunks with a roughly-equal sum. This algorithm returns the "breaks" (the indices at which the sequence should be split). Complexity: $O(N \cdot K)$.

```
1:  function DPOptimalEqualAreaBreaks(numbers)
2:      $AVG \leftarrow$ sum($numbers$) / $K$    ▷ Average chunk sum.
3:      $best\_error \leftarrow$ <2D array with $N+1$ rows (from 0 to
                      $N$) for the "end index" $m$,
                      and $K$ columns (from 0 to $K-1$) for
                      the number of "breaks" $b$.>
4:      $best\_breaks \leftarrow$ <2D array like above, for the breaks>

5:      for $m$ in 0..$N$ do  ▷ Fill the first column (column 0).
6:          $best\_error[m][0] \leftarrow |\text{PSum}(0, m) - AVG|$
7:          $best\_breaks[m][0] \leftarrow []$

8:      for $b$ in 1..$K-1$ do                    ▷ Loop over breaks.
9:          $m \leftarrow N$
10:         $break \leftarrow N$          ▷ The position of the last break.
11:         while $m \geq 0$ do
12:             if $break > m$ then
13:                 $break \leftarrow m$

14:             ▷ Go back until reaching the candidate positions
                for the last break:
15:             while (PSum($break$, $m$)< $AVG$ AND
                        $break > 0$) do
16:                 $break \leftarrow break - 1$

17:             ▷ Choose between the two candidates for the last
                break:
18:             if $best\_error[break+1][b-1]$
                    $+ |\text{PSum}(break+1, m) - AVG| <$
                    $best\_error[break][b-1]$
                    $+ |\text{PSum}(break, m) - AVG|$ then
19:                 $break \leftarrow break + 1$

20:             ▷ After choosing the better break, add it to the
                arrays.
21:             $best\_error[m][b] \leftarrow$ best_error$[break][b-1]$
                                $+ |\text{PSum}(\text{break, m}) - AVG|$
22:             $best\_breaks[m][b] \leftarrow best\_breaks[break][b-1]$
                                $+ [break]$
23:             $m \leftarrow m - 1$
24:     return $best\_breaks[N][K-1]$
```

2.4.6 Computational Complexity

Algorithm 4 fills a table with $O(N)$ rows and $O(K)$ columns, and takes constant time to fill each cell. Therefore, the total space and time complexity is $O(N \cdot K)$, which is almost linear, as K is usually low (3-10 colors on the map).

Note that filling each column (with N cells) takes $O(N)$ time, because as explained in Key Idea 3, the variables m and *break* which control the nested loops in lines 12 and 17 of the pseudocode, start from N and go backwards together until reaching 0. In other words, these two nested while loops run in linear time.

Technical note: As written in the pseudocode, the algorithm requires $O(N \cdot K^2)$ space, because best_breaks$[m][b]$ stores b breaks. However, this can be easily optimized such that best_breaks$[m][b]$ would store only the last break and a pointer to the list of the other breaks (stored in a different cell), etc., and recursively looking up all the b optimal breaks in the end. This way the space complexity will be $O(N \cdot K)$ as promised.

2.4.7 Caveat - Using a Different "Optimization Criteria"

At the start of this section, we defined the criterion for optimality to be the difference in absolute value from the average chunk sum. An alternative criterion could be the sum of squares of the differences. In this case, key idea 1 will no longer hold, as shifting the breaks has a different effect on the squared errors. Therefore, we cannot only consider two candidate locations for the last break. Instead, we will need to consider all the possible locations. That will require linear time for computing every cell in the two-dimensional array, leading to a total complexity of $O(N^2 \cdot K)$. However, in the case of map coloring, we do not think that the user will notice any difference when using a different criteria for optimality.

3. EVALUATION

3.1 Equal-area algorithms performance evaluation

To evaluate the performance of the above algorithms above and compare them with the naive approach of partitioning the list into equal-length chunks, we took the areas of 190 countries (area in pixels on the screen using the Winkel-Tripel projection [9]), in increasing order of their corresponding population.

[0.32, 0.1, 0.02, 0.21, 0.26, 5669.01, 0.18, 0.92, 0.58, 0.26, 0.54, 0.91, 0.56, 0.3, 0.35, 0.62, 1.11, 0.7, 4.08, 15.74, 0.51, 0.09, 197.86, 28.36, 15.99, 7.16, 0.26, 3.93, 4.45, 172.82, 36.41, 1.78, 30.99, 51.64, 252.56, 7.41, 0.72, 26.81, 26.05, 19.41, 21.22, 2.34, 78.28, 6.02, 303.18, 38.02, 13.66, 12.18, 707.48, 35.07, 29.17, 37.98, 1005.61, 107.11, 13.57, 2309.15, 384.04, 22.12, 38.84, 40.5, 90.29, 104.97, 1254.43, 231.63, 111.66, 48.66, 403.85, 73.33, 13.35, 394.82, 77.1, 96.79, 108.4, 726.11, 63.14, 88.62, 828.12, 0.62, 642.31, 147.84, 637.83, 72.1, 69.51, 278.39, 159.72, 83.93, 25.55, 293.15, 499.13, 113.85, 2028.47, 66.61, 626.53, 194.11, 157.26, 28.58, 140.46, 60.05, 732.25, 123.15, 31.79, 118.72, 136.38, 559.54, 824.47, 327.59, 135.87, 1332.76, 33.22, 60.0, 29.66, 287.38, 203.66, 118.02, 122.7, 46.86, 137.26, 1513.36, 176.04, 470.62, 233.95, 896.86, 229.25, 309.11, 135.05, 1504.34, 142.56, 1416.97, 4036.51, 322.35, 57.32, 1009.8, 1474.3, 543.28, 80.46, 724.24, 339.77, 376.25, 244.48, 952.56, 10515.05, 174.86, 546.75, 280.03, 2388.55, 625.66, 413.73, 1097.49, 848.21, 1573.78, 190.92, 568.91, 736.63, 2228.51, 283.84, 17333.09, 2911.42, 485.94, 3719.94, 697.32, 1111.94, 1373.61, 892.46, 690.44, 136.01, 1541.35, 846.82, 419.43, 383.9, 651.52, 799.1, 2728.16, 1056.83, 2114.65, 1250.89, 1337.73, 546.0, 420.43, 377.64, 2548.07, 525.77, 29575.45, 174.0, 1069.13, 1130.23, 10211.55, 2399.84, 13672.46, 3993.44, 12925.08]

We partitioned this list into 5 chunks (assuming 5 colors for example) using all the algorithms, and measured the average error for each. Table 2 summarizes the results.

Algorithm	Average Error
Naive equal-length (Quantiles)	34,928
Greedy algorithm 1	15,192
Greedy algorithm 2	5,572
Optimal dynamic programming	3,244

Table 2: Average error of different algorithms (Average difference between the average/optimal chunk sum and each chunk.)

We notice that in the given example, using the first greedy algorithm to assign the colors almost halves the error, and the second greedy algorithm shrinks it even more. In addition, using the optimal algorithm produces a significantly better result than all the others. Since the complexity of the optimal algorithm is good (almost linear), we would recommend using it instead of the simpler algorithms.

3.2 Visual Comparison

In addition to the numerical evaluation above, we generated map visualizations for two datasets (corresponding to two properties):

1. Population of countries
2. Population densities of countries

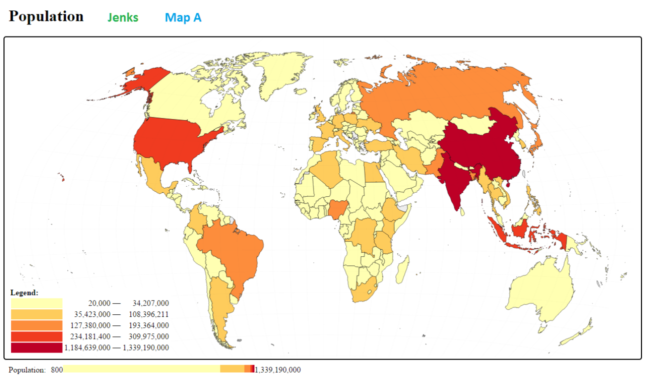
Note that the Wikipedia article on choropleth maps [1] states that a common error in choropleths is the use of raw data values (like population) to represent magnitude rather than normalized values to produce a map of densities. Regardless, we choose to test our algorithms on the population map, and that is because we design our systems such that any person can use (and not only experienced cartographers), and such mistakes are common in the viral maps that are becoming very popular on the web. Therefore, it is important for us to make sure that the coloring algorithm produces a good result even if the data is not completely suitable to be visualized on a choropleth map.

For each dataset, we compared three alternatives for assigning the colors to the regions:

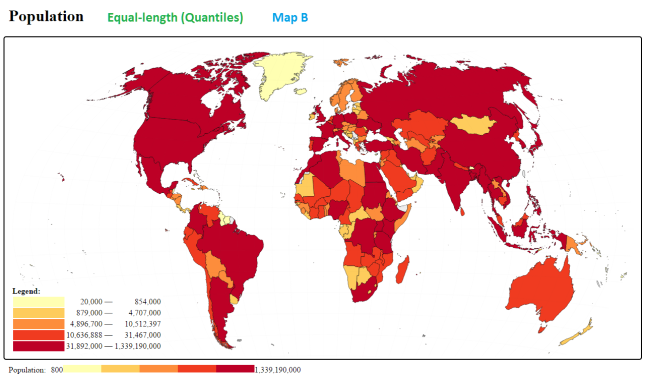
1. Jenks: The popular Jenks natural breaks optimization algorithm, which tries to cluster the regions into similarly-valued chunks.
2. Equal-length (Quantiles): The naive algorithm that assigns an equal number of regions to each color.
3. Equal-area: Our approach, which tries to obtain an evenly-colored map by assigning the ranges so that each color gets a roughly equal area on the screen.

Maps A, B, C, E, F, and G present every combination of dataset (population and population density) and algorithm (Jenks, equal-length, and equal-area).

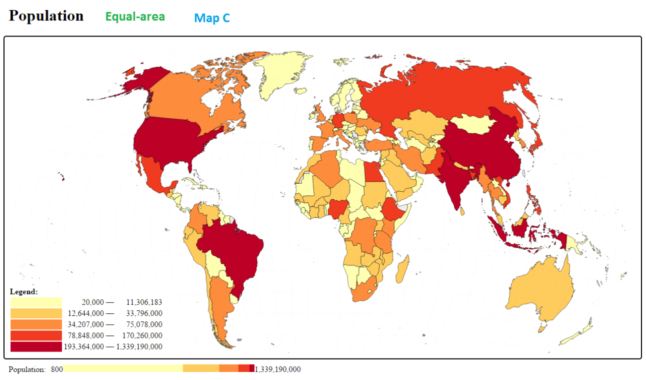
3.2.1 *Population Maps*



Map A: Jenks Natural Breaks
(Tries to cluster the regions into similarly-valued chunks.)

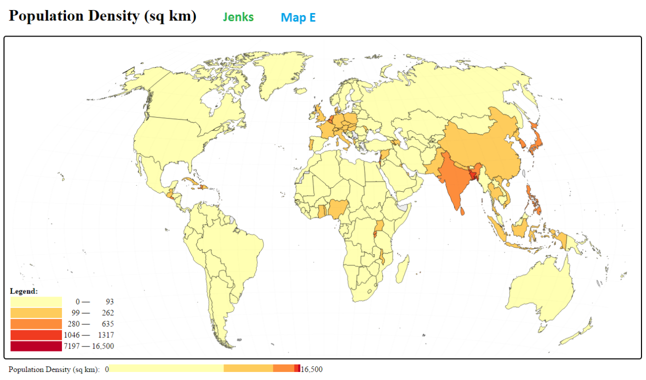


Map B: Equal-length (Quantiles)
(Each color is used for the same number of regions.)

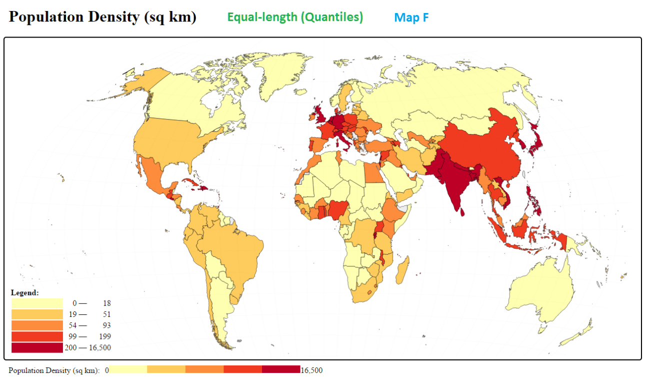


Map C: Equal-area
(Each color occupies roughly the same screen area.)

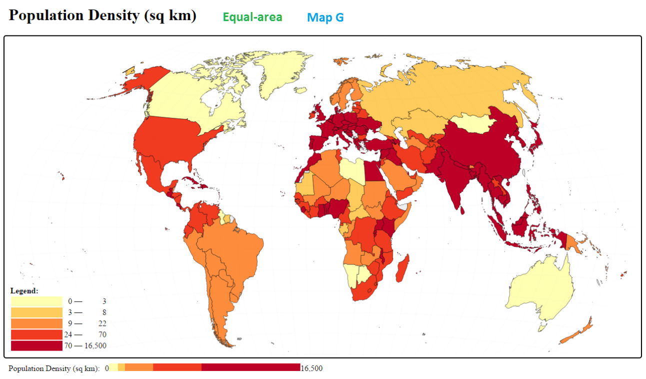
3.2.2 *Population Density Maps*


Population Density (sq km) Jenks Map E
Legend:
0 — 93
99 — 262
280 — 635
1046 — 1317
7197 — 16,500
Population Density (sq km): 0
16,500

Map E: Jenks Natural Breaks
(Tries to cluster the regions into similarly-valued chunks.)


Population Density (sq km) Equal-length (Quantiles) Map F
Legend:
0 — 18
19 — 51
54 — 93
99 — 199
200 — 16,500
Population Density (sq km): 0
16,500

Map F: Equal-length (Quantiles)
(Each color is used for the same number of regions.)


Population Density (sq km) Equal-area Map G
Legend:
0 — 3
3 — 8
9 — 22
24 — 70
70 — 16,500
Population Density (sq km): 0
16,500

Map G: Equal-area
(Each color occupies roughly the same screen area.)

3.2.3 *Notes*

From Maps A and E, we observe that the Jenks natural breaks algorithm does not differentiate well between the countries. This is especially the case for the population density map. On the one hand it makes it easy to identify the countries with a very high population density, and it has a nice and useful property that the bounds of each range in the legend are of the same order of magnitude. However, it is noticeable that the population density map is mostly light yellow, with some yellow, and barely any orange/red/brown, making the map look somewhat odd. Therefore, using an equal-area algorithm might be preferred in certain cases like this one. In addition, the equal-area algorithm is more efficient than Jenks in terms of computational complexity as we show below.

3.2.4 *Jenks Complexity*

An $O(K \times N \times log(N))$ algorithm is presented in [4], for the classification of an array of N numeric values into K classes such that the sum of the squared deviations from the class means is minimal, known as Fisher's Natural Breaks Classification. This algorithm is an improvement of Jenks' Natural Breaks Classification Method [14], which is a reimplementation of the algorithm described by Fisher [13] within the context of choropleth maps, which has time complexity $O(K \times N^2)$. We used a JavaScript implementation by Tom MacWright [6, 16].

4. OPTIMIZED AGORITHMS

We noticed that in Map G (density, equal-area), too many countries were colored in brown (more than half), covering almost all of Europe and Asia, leading to a perception that the map is not balanced even though the colors are equally distributed in terms of area.

This prompted us to improve our approach further, by taking into account the number of countries that each color is assigned to. I.e., we are trying to strike a balance between two things:

1. Each color covers roughly the same area (equal-area).
2. Each color is assigned to roughly the same number of regions (equal-length / quantiles).

4.1 Optimized Greedy Algorithm

In the first greedy algorithm that we discussed earlier, we kept adding to a chunk until it is "full" (i.e., its sum exceeded AVG–the average chunk sum). At this point, we make a few modifications to its so that we also try to achieve a balance of the lengths of the chunks and not only the sum of the areas of their elements. In particular, we keep inserting elements into a chunk until one of the following conditions is satisfied:

1. The chunk is full in terms of area (like before), and its length is at least half of average_length (a new condition to ensure that a color is not associated with too few countries).
2. The chunk is already of length 2 × average_length (a new condition to ensure that we don't have too many countries associated with the same color).
3. The chunk's area reached 2 × AVG (it's getting too big).

Note: As the chunks are not necessarily evenly-balanced, if we follow the conditions above naively, we might run out of elements before we get to the last chunk. We mitigate this issue by determining the breaks recursively. In particular, each time we find a break, we invoke the algorithm on the rest of the elements.

4.1.1 *Pseudocode*

The following pseudocode describes a recursive procedure to find the greedy breaks as described above. We invoke it using GetOptimizedGreedyBreaks(areas, 0, K) and it returns the indices of the K breaks.

Algorithm 5 Greedy algorithm for partitioning a sequence of N numbers into K chunks with roughly equal sum, while simultaneously trying to balance the lengths of the chunks as well. The result is something between equal-length (quantiles) and equal-area. This algorithm returns the "breaks" (the indices at which the sequence should be split). Complexity: $O(N)$.

```
1: U = 2      ▷ Upper bound on the chunk length/sum ratio. Chunk length and sum should not exceed twice those of the average chunk.
2: L = $\frac{1}{2}$      ▷ Lower bound on the chunk length ratio. Chunk length should be at least half the length of the average chunk.
3: function OptimizedGreedyBreaks($numbers$, $start\_index$, $num\_chunks$).
4:     if $num\_chunks == 1$ then
5:         return []                          ▷ One color - no breaks.
6:     $AVG \leftarrow \frac{\text{sum}(numbers)}{num\_chunks}$
7:     $AVG\_LEN \leftarrow \frac{numbers.\text{length} - start\_index}{num\_chunks}$
8:     $len \leftarrow 0$                    ▷ The length of the new chunk.
9:     $s \leftarrow 0$                      ▷ The sum of the new chunk.
10:    for $i = start\_index$ up to $numbers$.length do
11:        $len \leftarrow len + 1$
12:        $s \leftarrow s + numbers[i]$
13:        if ($s \geq AVG$ AND                         ▷ Condition 1
                $len \geq L \times AVG\_LEN$) OR
               ($len \geq U \times AVG\_LEN$) OR        ▷ 2
               ($s \geq U \times AVG$) then             ▷ 3
14:            ▷ Chunk is "full". Return the new break, plus the recursively-found other breaks.
15:            return [i + 1] + GetOptimizedGreedyBreaks($numbers$, $i+1$, $num\_chunks-1$)
```

4.2 Optimized Optimal Algorithm

Recall that when we discussed equal-area, we optimized: $ERROR\,(breaks) = \frac{1}{K} \cdot \sum_{i=1}^{K} |sum\,(chunk_i) - AVG|$. We now we modify this expression to account for the lengths of the chunks as well. We aim to minimize the following formula.

$$(1-W) \cdot \sum_{i=1}^{K} \left| \frac{\text{sum}(chunk_i) - AVG}{K \cdot AVG} \right|^2 + W \cdot \sum_{i=1}^{K} \left| \frac{\text{len}(chunk_i) - \frac{N}{K}}{N} \right|^2$$

where W is a user-defined constant specifying the weight to be given to the lengths. We use the term *W-score* to describe it on account of its similarity to the concept of an *f-score* used in information retrieval to vary the importance of precision and recall [18]. The left term corresponds to the normalized average error in area, while the right term corresponds to the normalized average error in length. Setting $W = 0$ would result in equal-area, and setting $W = 1$ would result in equal-length (quantiles). Setting $W = 0.5$ would result in some balance between area and length, which is what we are looking for.

4.2.1 Approach

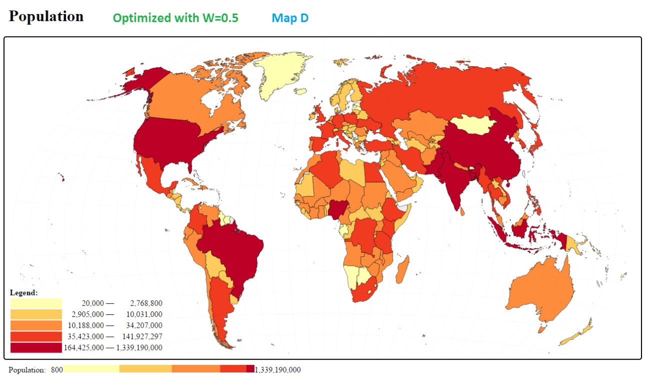
Again, we use dynamic programming to optimize the new criterion. However, the trick that we used to find only two candidate locations for the last break will not work here, as moving the break to the left or the right results not only in changing the sums of the chunks, but also their lengths. To overcome this, we consider all the locations for each break, at the cost of a higher complexity: $O(N^2 \times K)$.

4.2.2 Pseudocode

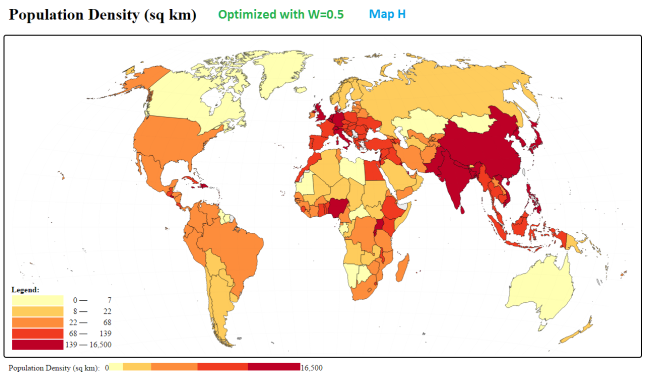
Algorithm 6 Dynamic programming optimal algorithm for partitioning a sequence of N numbers into K chunks with roughly equal sum, while simultaneously balancing the lengths of the chunks. The result is something between equal-length (quantiles) and equal-area. W specifies the desired weight to be given to the lengths in this optimization. This algorithm returns the "breaks" (the indices at which the sequence should be split). Complexity: $O(N^2 \cdot K)$.

```
1:  function DPOptimizedOptimalBreaks(numbers).
2:    $AVG \leftarrow$ sum($numbers$) / $K$        ▷ Average chunk sum.
3:    $AVG\_LEN \leftarrow N/K$                    ▷ Average chunk length.
4:    $best\_error \leftarrow$ <2D array with N+1 rows (from 0 to N)
                     for the "end index" m,
                     and K columns (from 0 to K-1) for the
                     number of "breaks" b.>
5:    $best\_breaks \leftarrow$ <2D array like above, for the breaks>

6:    for $m$ in $0..N$ do        ▷ Fill the first column (column 0).
7:      $best\_error[m][0] \leftarrow (1-W) \cdot \left|\frac{\textsc{PSum}(0,m) - AVG}{\textsc{PSum}(0,N)}\right|^2 + W \cdot \left|\frac{m - AVG\_LEN}{N}\right|^2$
8:      $best\_breaks[m][0] \leftarrow []$

9:    for $b$ in $1..K-1$ do                       ▷ Loop over breaks.
10:     for $m$ in $0..N$ do                       ▷ Loop over end-indices.
11:       $min\_error \leftarrow$ uninitialized
12:       $best\_break \leftarrow$ uninitialized
13:       ▷ Loop over candidate locations for break $b$.
14:       for $break$ in $0..m$ do
15:         ▷ Compute the error if we break at this index.
16:         $break\_error \leftarrow best\_error[break][b-1] + (1-W) \cdot \left|\frac{\textsc{PSum}(break,m) - AVG}{\textsc{PSum}(0,N)}\right|^2 + W \cdot \left|\frac{(m - break) - AVG\_LEN}{N}\right|^2$
17:         ▷ If this is the smallest error we have seen so
              far, update $min\_error$ and $best\_break$.
18:         if ($min\_error$ is uninitialized OR
                $break\_error < min\_error$) then
19:           $min\_error \leftarrow break\_error$
20:           $best\_break \leftarrow break$
21:       ▷ After choosing the best break, add it to the
            arrays.
22:       $best\_error[m][b] \leftarrow min\_error$
23:       $best\_breaks[m][b] \leftarrow best\_breaks[best\_break][b$ - 1$]+$
                                   $[best\_break]$
24:   return $best\_breaks[N][K-1]$
```

4.2.3 Visual Result



Map D: Population using Optimized algorithm ($W = 0.5$) Tries to balance between equal-area and equal-length.



Map H: Density using Optimized algorithm ($W = 0.5$) Tries to balance between equal-area and equal-length.

4.2.4 User Study

To evaluate our approach, we created two surveys (one for Population, comparing maps A, B, C, and D, and one for Population Density, comparing maps E, F, G, and H). Each survey presented the 4 maps, and after each map, two questions to ensure that users take a thorough look at each map:

1. How well do you think the color assignment on the map reflects the different population [density] ranges? Poorly / Average / Well

2. How well does the map represent the difference in population [density] of neighboring countries? Poorly / Average / Well

Finally, we ask users to choose the best map, the second best map, and the worst map. Using this information, we are able to infer their ranking of the 4 maps.

We asked 20 arbitrary people on Amazon Mechanical Turk to fill each survey. Tables 3 and 4 summarize the rankings.

	#1	**#2**	**#3**	**#4**
Jenks	7	2	7	4
Equal-length	0	1	3	16
Equal-area	5	10	5	0
Optimized	8	7	5	0

Table 3: User study results for population

	#1	#2	#3	#4
Jenks	2	1	2	15
Equal-length	7	5	8	0
Equal-area	5	2	10	3
Optimized	6	12	0	2

Table 4: User study results for population density

These results are fairly consistent with an earlier survey we performed with 14 random friends, on similar maps but using the Mercator projection (before we switched to the Winkel-Tripel projection). Tables 5 and 6 summarize the results from that survey.

	#1	#2	#3	#4
Jenks	2	1	7	4
Equal-length	2	1	3	8
Equal-area	2	8	3	1
Optimized	8	4	1	1

Table 5: User study results for population in older survey using Mercator projection

	#1	#2	#3	#4
Jenks	1	1	1	11
Equal-length	2	9	2	1
Equal-area	2	3	7	2
Optimized	9	1	4	0

Table 6: User study results for population density in older survey using Mercator projection

We can aggregate these results by giving an algorithm 3 points for ranking first, 2 points for ranking second, 1 point for ranking third, and no points for ranking last. Table 7 shows the aggregate number of points each algorithm receives based on the user rankings in tables 3-6.

	Winkel-Tripel		Mercator		Overall
	Pop.	Den.	Pop.	Den.	Overall
Jenks	32	10	15	6	63
Equal-length	5	39	11	26	81
Equal-area	40	29	25	19	113
Optimized	43	42	33	33	151

Table 7: Aggregate user study results (total number of points per algorithm) - 34 users total

From the results, we observe that the Optimized algorithm was overall preferred more than the others. It collected the largest number of points in the aggregate results for all 4 use-cases, and was ranked #1 by 46% of the users (compared to 16-21% for the other algorithms).

We understand that this user study is fairly simple, featuring only two datasets (population and population density for countries), and a small number of users. In addition, we understand that choropleth maps also have a competing objective of conveying information and not merely to look visually pleasing. However, both the visual quality and the information content of a map are somewhat subjective measures, and different algorithms appeal to different people. Nevertheless, we think that the equal-area and the Optimized algorithms should be considered as strong candidates when automatically assigning ranges to colors. When implementing the Optimized algorithm, we suggest having a slider that can be used to evaluate the effect of varying W. We leave it to the map compilers to decide which algorithm best fits their dataset.

5. CONCLUDING REMARKS AND FUTURE WORK

5.1 Further User Studies

It is worth noting that generally-speaking, larger countries (area-wise) tend to have larger populations and lower densities. Since the equal-length algorithm assigns the colors such that there is an equal number of countries in each bucket, many large countries fall into the darker buckets in the population case, and many large countries fall into the lighter buckets in the density case, resulting in a fairly dark Population map (map B), and a fairly light Density map (map F). Users seemed to like the lighter map, but not the darker one, which leads us to think that people prefer maps that are dominated by light colors rather than dark colors. This theory needs to be tested in future studies. This could be done by observing how users' preferences change when flipping the colors on the map, or using a different color set. It is also interesting to research whether changing the color of the borders between regions from black to white has any effect on the results.

Also, instead of providing users with several maps, asking to rank them, we can alternatively provide an interactive map that is colored using the Optimized algorithm and offers a dynamic slider to change the value of W. This would compress, in one interactive map, the entire range of maps from equal-area (with $W = 0$), to equal-length (with $W = 1$), passing through $W = 0.5$ we used in maps D and H. It would be interesting to see how users choose to set the value of W in each case.

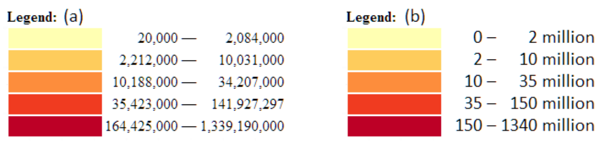
5.2 New Algorithms

The feedback we received also revealed that some users prefer more variability (contrast) between the colors of adjacent countries in order to facilitate the easier discovery of differences. This reinforces the motivation for the work on the classical four color map labeling problem [5]. In particular, the criteria that we discussed in this paper could also be modified to minimize the number of adjacent regions with the same color.

We also have reason to believe that people like to be able to easily identify the extremes on the map (e.g., "most dense country"). To address that, we can come up with an algorithm that assigns the colors on some kind of a bell curve, such that fewer regions fall in the first and last color ranges.

Another direction is to find some middle ground between the equal-area/equal-length algorithms that ignore the values when partitioning, and the clustering algorithms (like Jenks) which focus on finding natural breaks in the values.

Finally, a number of users expressed a preference for nice round numbers in the legend, rather than algorithmically-chosen ones which were too specific, and thus once we determine the break points, we could shift them slightly to be more user-friendly. For example, users prefer the legend in Figure 5b to the one in Figure 5a. Note however, that simply shifting the numbers that were chosen by the algorithm could result in changing the coloring of the map, so this needs to be done carefully.



Legend: (a)

🟨	20,000 —	2,084,000
🟧	2,212,000 —	10,031,000
🟧	10,188,000 —	34,207,000
🟥	35,423,000 —	141,927,297
🟥	164,425,000 —	1,339,190,000

Legend: (b)

🟨	0 –	2 million
🟧	2 –	10 million
🟧	10 –	35 million
🟥	35 –	150 million
🟥	150 –	1340 million

Figure 5: (a) Legend generated by our algorithms versus (b) a more user-friendly legend.

6. REFERENCES

[1] Choropleth Map. `http://en.wikipedia.org/wiki/Choropleth_map`.

[2] Color Brewer 2.0: Color Advice for Cartography. `http://colorbrewer2.org`.

[3] D3 Javascript Library. `http://d3js.org`.

[4] Fisher's Natural Breaks Classification. `http://wiki.objectvision.nl/index.php/Jenks_Natural_Breaks_Classification`.

[5] Four color theorem. `http://en.wikipedia.org/wiki/Four_color_theorem`.

[6] Implementation of Jenks Natural Breaks Algorithm. `https://gist.github.com/tmcw/4977508`.

[7] South America. `http://en.wikipedia.org/wiki/South_America`.

[8] VisuMaps - Visualize Data on Maps. `http://www.visumaps.com`.

[9] Winkel tripel projection. `http://en.wikipedia.org/wiki/Winkel_tripel_projection`.

[10] M. R. C. Coulson. In the matter of class intervals for choropleth maps: with particular reference to the work of George F. Jenks. *Cartographica*, 24(2):16–39, 1987.

[11] I. David M. Goldberg, J. Richard Gott. Flexion and Skewness in Map Projections of the Earth. *Cartographica*, 42(4):297–318, 2007.

[12] I. S. Evans. The selection of class intervals. *Transactions of the Institute of British Geographers*, 2:98–124, 1977.

[13] W. D. Fisher. On grouping for maximum homogeneity. *American Statistical Association Journal*, 534:789–798, 1958.

[14] G. F. Jenks. Optimal data classification for choropleth maps. Geography Department Occasional Paper No. 2 2, University of Kansas, Lawrence, KS, 1977.

[15] B. Jiang. Head/tail breaks: A new classification scheme for data with a heavy-tailed distribution. *The Professional Geographer*, 65(3):482–494, 2013.

[16] T. MacWright. Literate Jenks Natural Breaks and How The Idea Of Code is Lost. `http://www.macwright.org/2013/02/18/literate-jenks.html`.

[17] A. H. Robinson, J. L. Morrison, P. C. Muehrcke, J. Kimerling, and S. C. Guptill. *Elements of Cartography*. Wiley, New York, sixth edition, 1995.

[18] G. Salton. *Automatic Text Processing: The Transformation Analysis and Retrieval of Information by Computer*. Addison-Wesley, Reading, MA, 1989.

[19] W. Tobler. Thirty-five years of computer cartograms. *Annals of the Association of American Geographers*, 94:58–73, 2004.