# Squarified Treemap

## A Functional Implementation in *Mathematica*

By Jeffrey Starr, 2014-02

## Abstract

This paper presents an implementation of the Squarified Treemap algorithm (originally published by Bruls, Huizing, and van Wijk)[1] for a single-level of hierarchy written in a functional style using the *Mathematica* language.

## Introduction

A treemap is a popular visualization technique because of its density (high data content per square inch) and ability to handle hierarchical data. Treemaps visualize quantities by creating rectangles with areas that match (proportionally, at least) the original quantities. Viewers compare values by comparing areas. By using rectangles, treemaps are more dense than pie charts and there is evidence humans can compare areas of rectangles more accurately than wedges of a circle. Treemaps can also connote hierarchy by embedding rectangles in other rectangles recursively. This technique is often used to help viewers find files consuming excessive disk space by plotting the file size as the area of the rectangle and embedding the file rectangles within other rectangles to show the directory structure.[2]

## Rectangle, the Core Data Structure

The Squarified Treemap algorithm draws rectangles in other rectangles. Not all treemap algorithms use rectangles, but rectangles are the most popular choice.

*Mathematica* provides an existing data structure for a Rectangle which we will use within this paper.

In[1]:= `? Rectangle`

> Rectangle[$\{x_{min}, y_{min}\}, \{x_{max}, y_{max}\}$] is a two−dimensional
>    graphics primitive that represents a filled rectangle, oriented parallel to the axes.
>
> Rectangle[$\{x_{min}, y_{min}\}$] corresponds to a unit square with its bottom−left corner at $\{x_{min}, y_{min}\}$.  ≫

Note that x and y refer to Cartesian coordinates, not screen coordinates. Beyond this data structure, we need several utility functions:

```
In[5]:= Coord[side_, Rectangle[{xmin_, ymin_}, {xmax_, ymax_}]] :=
         Switch[side,
           Top, ymax,
           Bottom, ymin,
           Left, xmin,
           Right, xmax]
```

```
In[6]:= Width[r_Rectangle] := Abs[Coord[Right, r] - Coord[Left, r]]
```

```
In[7]:= Height[r_Rectangle] := Abs[Coord[Top, r] - Coord[Bottom, r]]
```

```
In[4]:= Area[r_Rectangle] := Width[r] × Height[r]
```

The aspect ratio of a rectangle is the proportion of the longest edge to the shorter edge (thus, the aspect ratio is always greater than or equal to one). For a degenerate rectangle (where either the height or width is zero), we will define the aspect ratio as infinite.

```
In[9]:= RectAspectRatio[r_Rectangle] :=
         Which[
           Width[r] == 0 ⋁ Height[r] == 0, ∞,
           Width[r] ≥ Height[r], Width[r] / Height[r],
           Width[r] < Height[r], Height[r] / Width[r]
         ]
```

## Examples: Rectangle Functions

```
In[13]:= Block[
           {r = Rectangle[{0, 0}, {6, 4}]},
           {Width[r], Height[r], Area[r], RectAspectRatio[r]}
         ]
```

$$Out[13]= \left\{6, 4, 24, \frac{3}{2}\right\}$$

# Sidebar: Visualization

The algorithm presented in this paper is independent of a graphical rendering system. However, as we generate rectangles, it is useful to see them plotted for verification. The function below takes a list of rectangles and plots them, with axes and a gridline, using *Mathematica*'s graphic capabilities.
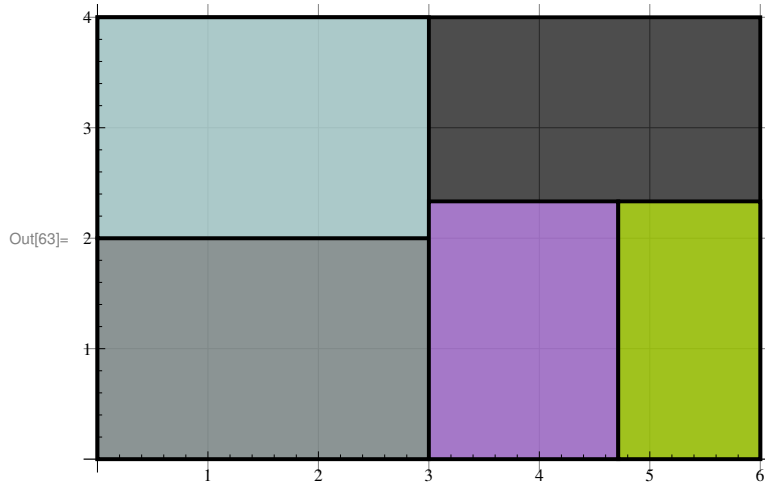
```
In[62]:= VisualizeRectangles[rectangles_List] :=
         Graphics[
           Join[
             {EdgeForm[{Thick, Black}],
               Opacity[7 / 10]},
             Riffle[rectangles, ColorData["Atoms", "ColorList"]]
           ],
           Axes → True,
           GridLines → Automatic
         ]
```

As an example of the output:

```
In[63]:= VisualizeRectangles[
    {Rectangle[{0, 0}, {6, 4}],
     Rectangle[{0, 0}, {3, 2}],
     Rectangle[{0, 2}, {3, 4}],
     Rectangle[{3, 0}, {33 / 7, 7 / 3}],
     Rectangle[{33 / 7, 0}, {6, 7 / 3}]}]
```

Out[63]=



# Filling a Rectangle

Given a rectangle and a list of desired areas for sub-rectangles (the sum of those areas is equal to or less than the area of the parent rectangle), we can iteratively fill the parent rectangle with child rectangles. For the squarified treemap algorithm, we fill in rectangles either vertically (stack children on top of each other, flushed to the left) or horizontally (stack children to the right of each other, flush to the left and right edges and growing upwards).

In the vertical case, the height of each rectangle is proportional and scaled to the total areas of the children and the fixed height of the parent. Contrawise, in the horizontal case, the widths of each rectangles are proportional and scaled to the fixed width of the parent. Within these constraints, the width or height (respectively) can be computed using the child's desired area.

## Fill Vertically

```
In[64]:= FillRectangle[parent_Rectangle, {}, Vertical] := {}
```

```
In[73]:= FillRectangle[parent_Rectangle, areas_List, Vertical] :=
      Block[
        {heights = Prepend[Height[parent] * Accumulate[areas] / Total[areas], 0]},
        Table[
         Rectangle[
          {Coord[Left, parent],
           Coord[Bottom, parent] + heights[[i]]},
          {Coord[Left, parent] + areas[[i]] / (heights[[i + 1]] - heights[[i]]),
           Coord[Bottom, parent] + heights[[i + 1]]}
         ],
         {i, Length[heights] - 1}
        ]
      ]
```

## Fill Horizontally

```
In[66]:= FillRectangle[parent_Rectangle, {}, Horizontal] := {}
```

```
In[72]:= FillRectangle[parent_Rectangle, areas_List, Horizontal] :=
      Block[
        {widths = Prepend[Width[parent] * Accumulate[areas] / Total[areas], 0]},
        Table[
         Rectangle[
          {Coord[Left, parent] + widths[[i]],
           Coord[Bottom, parent]},
          {Coord[Left, parent] + widths[[i + 1]],
           Coord[Bottom, parent] + areas[[i]] / (widths[[i + 1]] - widths[[i]])}
         ],
         {i, Length[widths] - 1}
        ]
      ]
```

## Examples: Fill Horizontally and Vertically

```
In[93]:= Block[
    {parent = Rectangle[{0, 0}, {6, 4}],
     p2 = Rectangle[{3, 0}, {6, 4}]},
    GraphicsRow[
     {VisualizeRectangles[Join[{parent}, FillRectangle[parent, {6}, Vertical]]],
      VisualizeRectangles[
       Join[{parent}, FillRectangle[parent, {6, 6}, Vertical]]],
      VisualizeRectangles[
       Join[{parent}, FillRectangle[parent, {6, 6, 4}, Vertical]]],
      VisualizeRectangles[
       Join[{parent}, FillRectangle[p2, {4}, Horizontal]]],
      VisualizeRectangles[
       Join[{parent}, FillRectangle[p2, {4, 3}, Horizontal]]]
     },
     ImageSize → Large,
     Spacings → 0
    ]
   ]
```
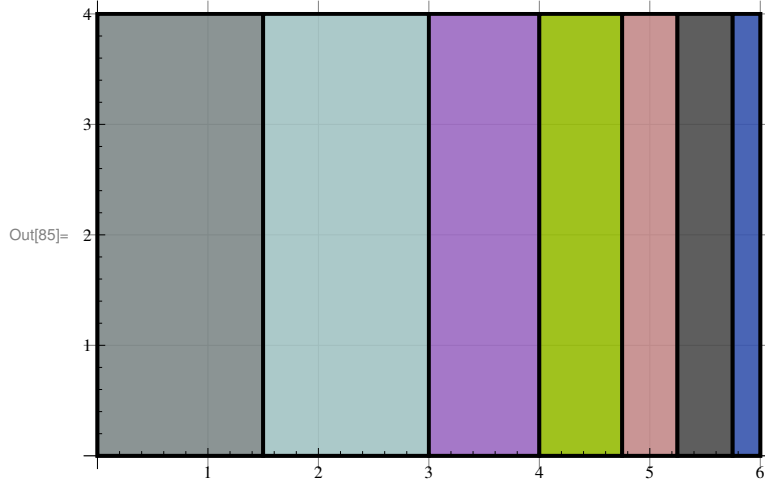
Out[93]=



---
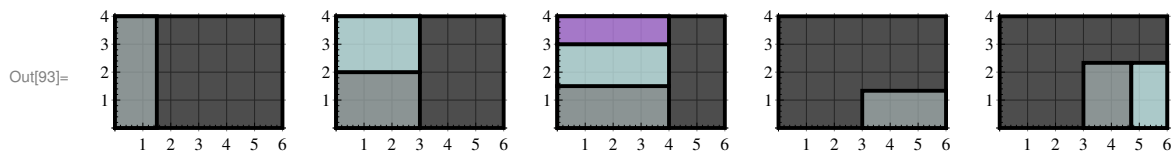
# How Many Children?

We can add all children to a rectangle using the same orientation. However, this leads to narrow rectangles that hard to compare:

In[85]:=
```
Block[
  {parent = Rectangle[{0, 0}, {6, 4}]},
  VisualizeRectangles[
   Join[{parent}, FillRectangle[parent, {6, 6, 4, 3, 2, 2, 1}, Horizontal]]
  ]
]
```

Out[85]=

The squarified treemap algorithm uses a heuristic to first fill a rectangle with children using one orientation, then switch to the opposite orientation, and so on. The algorithm places a child rectangles until the worst aspect ratio of the group of placed rectangles increases. In other words, rectangles are placed until a local minima of the aspect ratio is found. The worst aspect ratio is simply the maximum aspect ratio of a list. If the list is empty, we define the worst aspect ratio as infinite.

In[95]:=
```
WorstAspectRatio[{}] := ∞
```

In[94]:=
```
WorstAspectRatio[rectangles_List] :=
  Max[RectAspectRatio[#] & /@ rectangles]
```

In[96]:=
```
PlaceTillLocalMinima[parent_Rectangle, areas_List, orientation_] :=
  PlaceTillLocalMinima[parent, areas, orientation, {}]
```

The PlaceTillLocalMinima (4 argument) function takes a parent rectangle, a list of desired areas for child rectangles, an orientation (either Vertical or Horizontal), and accumulates a list of rectangles placed using FillRectangle.

In[97]:=
```
PlaceTillLocalMinima[parent_Rectangle, areas_List, orientation_, placed_List] :=
  If[Length[placed] == Length[areas], placed (* local minima achieved *),
   Block[
     {placeStep =
       FillRectangle[parent, Take[areas, Length[placed] + 1], orientation]},
     If[WorstAspectRatio[placeStep] > WorstAspectRatio[placed],
      placed(*previous step had local minima*),
      PlaceTillLocalMinima[parent, areas, orientation, placeStep]
      (*recurse and check if next step is local minima*)
     ]
    ]
  ]
```

When a local minima is found, the upper-level function will place the remaining children using the opposite orientation. This function provides the flip:

In[109]:= **InvertOrientation[Horizontal] := Vertical**

In[110]:= **InvertOrientation[Vertical] := Horizontal**

## Examples: Finding Local Minima

The function finds that placing just the first two rectangles reaches the local minima:

In[101]:= **PlaceTillLocalMinima[Rectangle[{0, 0}, {6, 4}], {6, 6, 4, 3, 2, 2, 1}, Vertical]**

Out[101]= {Rectangle[{0, 0}, {3, 2}], Rectangle[{0, 2}, {3, 4}]}

In[102]:= **WorstAspectRatio[%]**

Out[102]= $\dfrac{3}{2}$

The worst aspect ratio for the next step (placing 6, 6, and 4) is 4, which is greater than 3/2.

In[104]:= **WorstAspectRatio[FillRectangle[Rectangle[{0, 0}, {6, 4}], {6, 6, 4}, Vertical]]**

Out[104]= 4

Similarly, the function finds the local minima after the first two rectangles:

In[105]:= **PlaceTillLocalMinima[Rectangle[{3, 0}, {6, 4}], {4, 3, 2, 1, 1}, Horizontal]**

Out[105]= $\left\{\text{Rectangle}\left[\{3, 0\}, \left\{\dfrac{33}{7}, \dfrac{7}{3}\right\}\right], \text{Rectangle}\left[\left\{\dfrac{33}{7}, 0\right\}, \left\{6, \dfrac{7}{3}\right\}\right]\right\}$

In[106]:= **WorstAspectRatio[%]**

Out[106]= $\dfrac{49}{27}$

The worst aspect ratio for the next step (4, 3, and 2) is 9/2, which is greater than 49/27.

In[108]:= **WorstAspectRatio[FillRectangle[Rectangle[{3, 0}, {6, 4}], {4, 3, 2}, Horizontal]]**

Out[108]= $\dfrac{9}{2}$

# Calculating Remaining Dimensions

After the algorithm has placed a number of children, it will invert orientation and place the remaining children. The parent rectangle for the first group of children is different than the second group (and the third and fourth...) because some of the space has already been consumed. These pair of functions are used to calculate the "new" parent rectangle for the next iteration of rectangle placing.

First, given a group of placed rectangles, we need to find the bounding box around them. Due to the algorithm's design, the bounding box will always be a rectangle flush with the child rectangles.

```
In[111]:=  EnclosingRectangle[rectangles_List] :=
            Rectangle[
             {Min[Coord[Left, #] & /@ rectangles],
              Min[Coord[Bottom, #] & /@ rectangles]},
             {Max[Coord[Right, #] & /@ rectangles],
              Max[Coord[Top, #] & /@ rectangles]}
            ]
```

Second, the remaining rectangle is formed from the right edge of the child (if the children were placed Vertically) or from the top edge of the child (if the children were placed Horizontally). The top-right corner is always equal to the parent.

```
RemainingRectangle[parent_Rectangle, child_Rectangle] :=
 Rectangle[
  {If[Coord[Bottom, parent] == Coord[Bottom, child] ⋀
      Coord[Top, parent] == Coord[Top, child],
    Coord[Right, child],
    (*else*)Coord[Left, parent]],
   If[Coord[Left, parent] == Coord[Left, child] ⋀
      Coord[Right, parent] == Coord[Right, child],
    Coord[Top, child],
    (*else*)Coord[Bottom, parent]
   ]
  },
  {Coord[Right, parent],
   Coord[Top, parent]}
 ]
```

## Examples: Remaining Rectangle

```
In[116]:=  EnclosingRectangle[{Rectangle[{0, 0}, {3, 2}], Rectangle[{0, 2}, {3, 4}]}]
```

```
Out[116]=  Rectangle[{0, 0}, {3, 4}]
```

```
In[129]:=  RemainingRectangle[
            Rectangle[{0, 0}, {6, 4}],
            Rectangle[{0, 0}, {3, 4}]]
```

```
Out[129]=  Rectangle[{3, 0}, {6, 4}]
```

```
In[130]:=  RemainingRectangle[Rectangle[{3, 0}, {6, 4}],
```

$$\text{EnclosingRectangle}\left[\left\{\text{Rectangle}\left[\{3, 0\}, \left\{\frac{33}{7}, \frac{7}{3}\right\}\right], \text{Rectangle}\left[\left\{\frac{33}{7}, 0\right\}, \left\{6, \frac{7}{3}\right\}\right]\right\}\right]\right]$$

$$Out[130]=\ \text{Rectangle}\left[\left\{3, \frac{7}{3}\right\}, \{6, 4\}\right]$$

```
In[131]:=  RemainingRectangle[Rectangle[{0, 0}, {1, 1}], Rectangle[{0, 0}, {1, 1}]]
```

```
Out[131]=  Rectangle[{1, 1}, {1, 1}]
```

# Squarify Algorithm

With all the pieces together, we can now write the Squarify function. This function takes a parent rectangle and a list of areas (whose sum must equal the area of the parent) and returns a list of Rectangles.

```
In[132]:= Squarify[parent_Rectangle, areas_List] :=
           Squarify[parent, areas, {}, Vertical]
```

```
In[133]:= Squarify[parent_Rectangle, areas_List, placed_List, orientation_] :=
           If[Length[areas] == 0, placed(*done*),
            Block[
             {placeThisStep = PlaceTillLocalMinima[parent, areas, orientation]},
             Squarify[
              RemainingRectangle[parent, EnclosingRectangle[placeThisStep]],
              Drop[areas, Length[placeThisStep]],
              Join[placed, placeThisStep],
              InvertOrientation[orientation]
             ]
            ]
           ]
```

## Example: Squarify

```
In[134]:= Squarify[Rectangle[{0, 0}, {6, 4}], {6, 6, 4, 3, 2, 2, 1}]
```

$$\text{Out[134]=}\ \left\{\text{Rectangle}[\{0, 0\}, \{3, 2\}], \text{Rectangle}[\{0, 2\}, \{3, 4\}], \text{Rectangle}\left[\{3, 0\}, \left\{\frac{33}{7}, \frac{7}{3}\right\}\right],\right.$$

$$\text{Rectangle}\left[\left\{\frac{33}{7}, 0\right\}, \left\{6, \frac{7}{3}\right\}\right], \text{Rectangle}\left[\left\{3, \frac{7}{3}\right\}, \left\{\frac{21}{5}, 4\right\}\right],$$

$$\left.\text{Rectangle}\left[\left\{\frac{21}{5}, \frac{7}{3}\right\}, \left\{6, \frac{31}{9}\right\}\right], \text{Rectangle}\left[\left\{\frac{21}{5}, \frac{31}{9}\right\}, \{6, 4\}\right]\right\}$$

```
In[135]:= VisualizeRectangles[%]
```
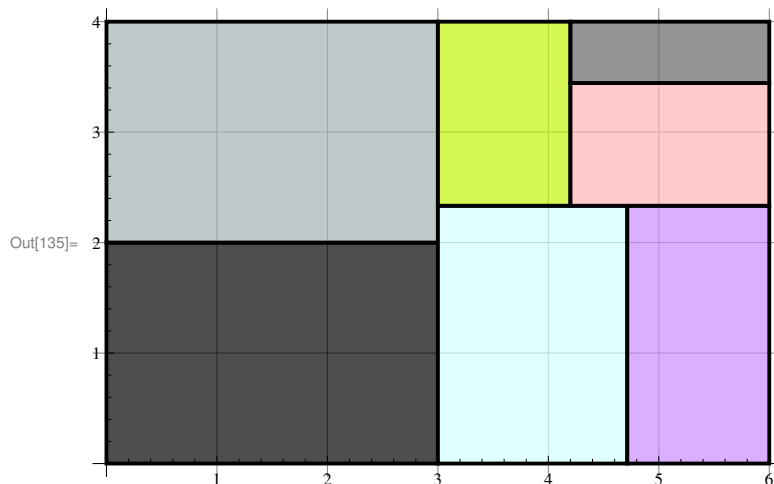
# Dealing with User Data

The previous functions relied on the fact that the area of the parent rectangle was greater than or equal to the sum of the areas. User data, however, will typically need to be scaled. *Treemap* is a wrapper function that performs this scaling and returns a graphic object generated using VisualizeRectangles for largely arbitrary data. (This function also sorts the values in descending order. According to the original paper, this produces the best results.)

```
In[138]:=  Treemap[width_Integer, height_Integer, values_List] :=
  Block[
    {safeValues = Sort[Select[values, # > 0 &], Greater],
     parentArea = width * height},
    VisualizeRectangles[
     Squarify[
      Rectangle[{0, 0}, {width, height}],
      safeValues * (parentArea / Total[safeValues])
     ]
    ]
  ]
```

## Example: Treemap

```
In[139]:=  Treemap[320, 240, {858, 1020, 1119, 715, 976, 857, 900}]
```

Out[139]=
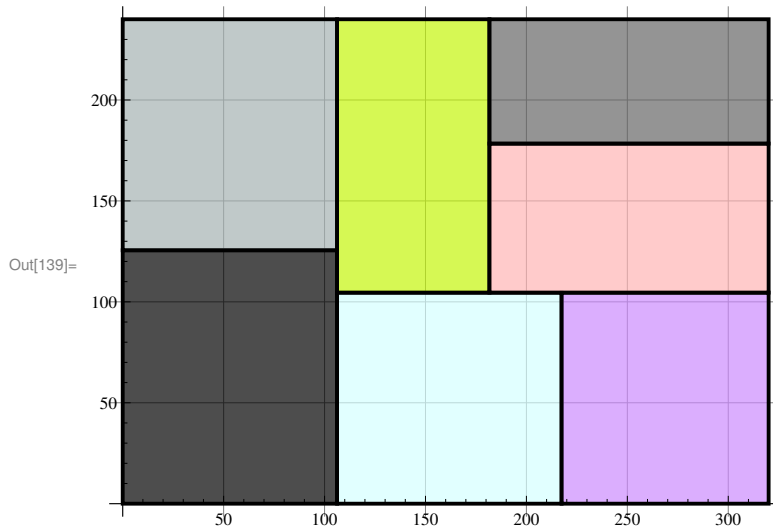


# References

1   M. Bruls, K. Huizing, and J. van Wijk, "Squarified treemaps," in Proceedings of the Joint Eurographics and IEEE TCVG Symposium on Visualization, pp. 33-42, 2000.

2    http://en.wikipedia.org/wiki/File:Tree_Map.png