Area-Proportional Venn Diagrams for D3

Helen Lu
Stanford University
helenlu@stanford.edu

ABSTRACT

D3 is a JavaScript library for creating visualizations on the Web. This paper describes a D3 extension that generates area-proportional Euler diagrams from client-provided data. The library calculates the optimal sizes and positions of circles in the diagram, which the client can display with existing D3 functionality. The implementation is based on Leland Wilkenson's gradient descent algorithm.

Author Keywords

d3 venn euler diagram

INTRODUCTION

Venn diagrams are a popular visualization type, easily understood by a wide audience. Traditionally, a Venn diagram simply shows all possible combinations of sets, and area is unimportant.

In area-proportional Venn diagrams, the area covered by a circle represents the relative size of the corresponding set, and the area covered by the intersection of circles represents the relative size of the intersection between those sets. The shapes used in these diagrams need not be circles, but this implementation uses only circles. Circles simplify the mathematical computation and look visually pleasing. Area-proportional Venn diagrams are useful when the goal is to visualize the relative sizes and overlaps of groups.

In this paper, I use the term "Venn diagram" interchangeably with the term "Euler diagram." Venn diagrams are one type of Euler diagrams. In reality, the visualizations discussed here are Euler diagrams, since not every possible intersection is required to be present.

D3 is a JavaScript library for creating visualizations on the Web [1]. Its existing functionality already allows clients to generate circles, color them, and place them anywhere on the screen. However, it is nontrivial to calculate the positions for circles in an area-proportional Venn diagram. This library bridges that gap by calculating the sizes and positions for circles, given the client's data. I use Wilkenson's algorithm for calculating area-proportional circular Venn/Euler diagrams [6].

It is not generally possible to generate a completely correct area-proportional Euler diagram when there are more than two sets. This D3 library function returns an approximation. The library also provides the client with a stress measure, which indicates goodness of fit. If the stress is unacceptably high, the client may decide to use a different visualization type.

RELATED WORK

Leland Wilkinson developed a gradient descent algorithm to calculate the locations and sizes of circles in an area-proportional circular Euler diagram. I use this algorithm in this D3 extension. Wilkinson has already implemented his algorithm in an R package called `venneuler` [5]. The source code for this R package is available online, under the Mozilla Public License. There are currently no other implementations of Wilkinson's method.

Wilkinson's R package is useful for certain audiences, but it cannot create the dynamic interaction that D3 enables. A D3 version can bring animated or interactive area-proportional Venn diagrams to a wide audience, through the Web.

Wilkinson describes other existing algorithms for generating area-proportional Euler diagrams, but argues that none provide approximations as good as his own [6].

Other software is available for generating static area-proportional Venn diagrams. Two examples are BioVenn [2] and eulerAPE [4], both of which can only handle up to 3 sets. Figure 1 and Figure 2 show screenshots of these applications, respectively.

METHODS

API Design

The client sees three new functions:

- `d3.layout.venn()`: Constructs a new Venn diagram layout.
- `venn.size()`: Returns the width and height of the visualization in a two-element array, when called with no arguments. Sets the width and height of the visualization when called with one argument, an array containing the width and height.

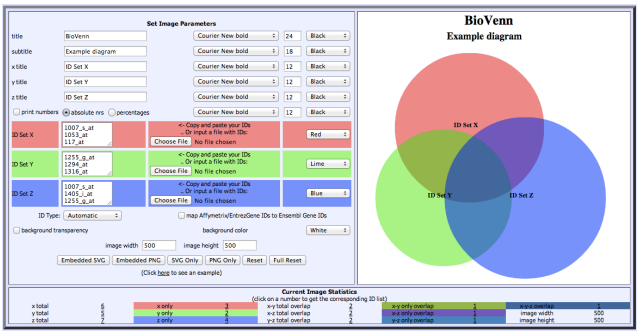

Figure 1. BioVenn screen capture.

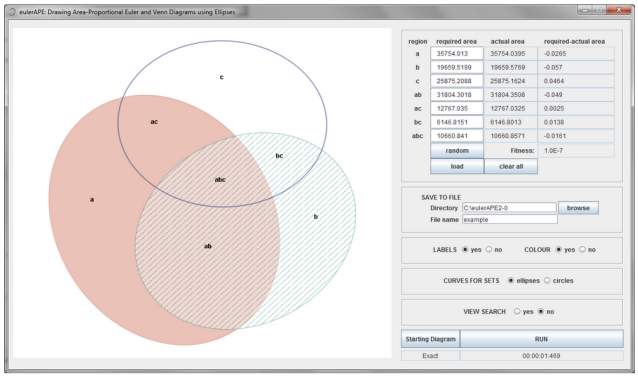

Figure 2. eulerAPE screen capture.

- `venn.stress()`: Returns the stress measure of the Venn diagram. Higher stress corresponds to poor fit.

The client can input data in two ways.

1. An array of arrays of strings. A single data point becomes an array of the sets it belongs to. For example, to construct a Venn diagram of online friend groups, one person would become an array that contains the groups he or she belongs to. The visualization in Figure 5 was created by this approach.

2. A single array of integers. Each integer represents the size of one region in the Venn diagram. Since a Venn diagram of n circles can divide the plane into up to 2^n regions, the length of this array must be equal to 2^n. The sets are numbered $0, 1, ...n-1$. The size of region A appears at index `1<<A`; note that this does not include portions of Set A that overlap with other sets. The size of the region $A \cap B$ appears at index `(1<<A) + (1<<B)`, etc. The visualization in Figure 6 was created by this approach. Figure 4 shows the sizes of each region.

A	B	$A \cap B$	C	$A \cap C$	$B \cap C$	$A \cap B \cap C$
001	010	011	100	101	110	111

Figure 3. Bitpatterns for Venn diagram segments.

Figure 4. Areas of Venn diagram regions.

If the client inputs data with the first representation, the library internally converts the data to the second representation. The first is perhaps more intuitive for the client, but the second is faster, particularly for larger datasets. Both are provided as options, because either one could be more suitable, depending on the use case.

The library computes the Venn diagram layout after data is entered. The client will have the following attributes available for each circle:

- `x`: x coordinate of the center.
- `y`: y coordinate of the center.
- `outerRadius`: Radius of the circle.
- `label`: String for the label. Available only when using input method #1.
- `xLabel`: Suggested x coordinate of the label.
- `yLabel`: Suggested y coordinate of the label.

Following D3 convention, the client must actually create circles and labels after entering data. The client must also choose colors, set reasonable opacity, etc. This library simply provides the sizes and positions of these elements.

The suggested coordinates of the label are located at the centroid of the region belonging only to the label's set. When this region's size is below a threshold, the center of the circle is used instead. This is a simple solution that sometimes fails. (See "Friends" in Figure 5.) The client is free to use or ignore these suggested positions.

The client should check `venn.stress()` after entering data. If stress is high, the approximate area-proportional di-

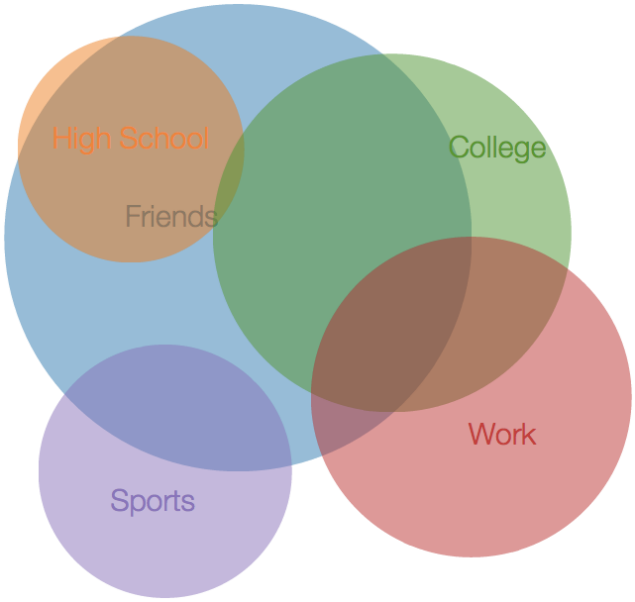

Figure 5. Area-proportional Euler diagram, using first data input method.

agram is not a good fit for the data. This means that the data's overlaps are impossible to show accurately in a circular area-proportional Venn diagram. In this case, the client should consider using a different type of visualization.

Client Use

The following code uses the first method of data input. Here, the client is creating one row for each person in his social network. The row contains the names of the social groups that the person belongs to. Thus `data` becomes an array of arrays, but not a 2D matrix. The length of the rows can vary. He creates three friends who belong to no other groups, three friends who are also in his high school group, two friends who are also in his college group, etc. The resulting visualization can be seen in Figure 5.

```
var A="Friends", B="High School", C="College";
var D="Work", E="Sports";
var groups = [A, B, C, D, E];

var data = [];
for (var i=0; i<3; i++) data.push([A]);
for (var i=0; i<4; i++) data.push([A, B]);
for (var i=0; i<6; i++) data.push([A, C]);
for (var i=0; i<1; i++) data.push([C]);
for (var i=0; i<2; i++) data.push([C, D]);
for (var i=0; i<1; i++) data.push([A, D]);
for (var i=0; i<1; i++) data.push([A, C, D]);
for (var i=0; i<4; i++) data.push([D]);
for (var i=0; i<3; i++) data.push([E]);
for (var i=0; i<2; i++) data.push([A, E]);
```

The code below uses the second method of data input. In this case, the client already knows the desired size of each region on the Venn diagram. He wishes to create a Venn diagram with 6 circles, so `data` contains 2^6 elements. He first sets all elements in `data` to zero, because most of the possible intersections do not occur. Then he individually sets the sizes of the intersections that do occur, using bitshifts and bitwise ORs to find the array indices. Figure 4 shows the areas in each region. The resulting visualization can be seen in Figure 6.

```
var data = [];
for (var i=0; i<Math.pow(2, 6); i++)
    data[i] = 0;

var A=0, B=1, C=2, D=3, E=4, F=5;
data[1<<A] = 4; data[1<<B] = 6; data[1<<C] = 3;
data[1<<D] = 2; data[1<<E] = 7; data[1<<F] = 3;
data[1<<A|1<<B] = 2; data[1<<A|1<<F] = 2;
data[1<<B|1<<C] = 2; data[1<<B|1<<D] = 1;
data[1<<B|1<<F] = 2; data[1<<C|1<<D] = 1;
data[1<<D|1<<E] = 1; data[1<<E|1<<F] = 1;
data[1<<A|1<<B|1<<F] = 1;
data[1<<B|1<<C|1<<D] = 1;
```

After the data is ready, the code to create the visualization is familiar to any D3 programmer. When `venn.size()` is used as a setter, it returns the same thing as `d3.layout.venn()`. This allows function calls to be chained in D3 style.

The client creates circles and sets their positions and radii according to the data generated by `d3.layout.venn()`. The colors come from the standard D3 color palette. The client should set opacity to less than one, since Venn diagrams have occlusion by definition.

The client generates text elements for labels, setting their text values to `d.label`. `d.label`, shown below, is available only if the first method was used for data input. If the second method was used, the client should have the label strings in a separate array. Then the label will be `labels[i]`, where `labels` is the name of this array.

Finally, the client sets the label x and y positions, which are provided relative to the circle centers.

```
var color = d3.scale.category10();
var venn = d3.layout.venn().size([800, 600]);

var circle = d3.svg.arc()
  .innerRadius(0)
  .startAngle(0).endAngle(2*Math.PI);

var vis = d3.select("body")
  .append("svg").data([data])
  .attr("width", 800).attr("height", 600);

var circles = vis.selectAll("g.arc")
  .data(venn).enter().append("g")
  .attr("class", "arc")
  .attr("transform", function(d, i){
    return "translate("+d.x+","+d.y+")"; });
circles.append("path")
  .attr("fill", function(d, i){return color(i);})
  .attr("opacity", 0.5)
  .attr("d", circle);
circles.append("text")
  .attr("text-anchor", "middle")
  .text(function(d, i){ return d.label; })
  .attr("fill", function(d, i){return color(i);})
```

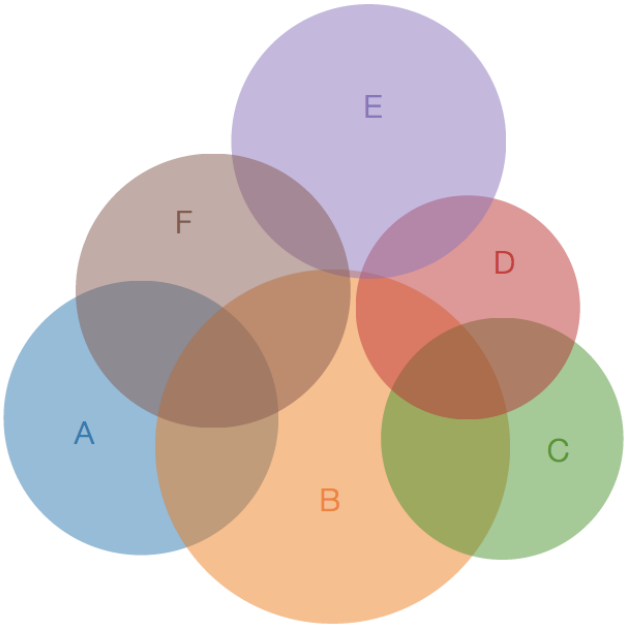

Figure 6. Area-proportional Euler diagram, using second data input method. Start of animation.

```
  .attr("x", function(d, i){ return d.labelX; })
  .attr("y", function(d, i){ return d.labelY; });
```

If the client wishes to launch an animation, he must first change the data. Currently, the client cannot add circles, although he may set existing circles to have zero size. When the following data changes are applied to the visualization in Figure 6, the result will be Figure 7.

```
data[1<<B] = 4; data[1<<E] = 1;
data[1<<A|1<<B|1<<F] = 3;
data[1<<B|1<<C|1<<D] = 2;
```

The following code is standard D3 for creating a transition after the data has changed. There are three sets of attributes changing:

1. The circle radii.
2. The circle centers.
3. The label centers.

```
vis.selectAll("path")
  .data(venn).transition().duration(2000)
  .attr("d", circle);
vis.selectAll("g.arc")
  .data(venn).transition().duration(2000)
  .attr("transform", function(d, i){
    return "translate("+d.x+","+d.y+")"; });
vis.selectAll("text")
  .data(venn).transition().duration(2000)
  .attr("x", function(d, i){ return d.labelX; })
  .attr("y", function(d, i){ return d.labelY; });
```

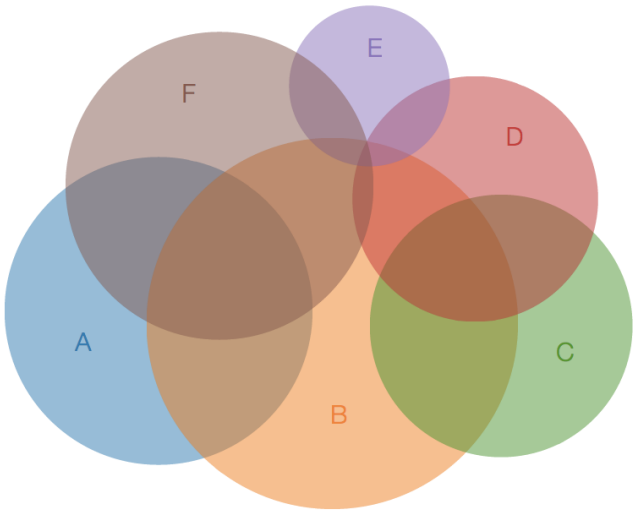

Figure 7. Area-proportional Euler diagram, using second data input method. End of animation.

The gradient descent does not start over when data is updated. Instead, it starts from the existing configuration. This strategy is potentially faster when the data changes are slight. More importantly, however, it ensures that the result will be suitable for a smooth animation. Figures 6 and 7 show that the circles remain in nearly the same positions. If the gradient descent starts over, the circles will likely end up in very different places, and the animation will look like circles flying across the screen.

By starting from the existing configuration, we risk getting caught in a local minima. If the client wishes to update data but does not intend to show a transition, he should create a new instance of `d3.layout.venn` instead of updating the existing one.

Algorithm

The implementation uses Wilkenson's method for finding approximate area-proportional diagrams [6]. Wilkenson uses gradient descent to minimize the error between the desired areas of each segment and the actual areas. The gradient descent can get caught in local minima, but this is less likely when good starting locations are chosen. Wilkenson describes a method for choosing these starting locations, using singular value decomposition. I choose starting locations this way when data is first entered.

When data is updated, I skip this step, which allows the circles to remain in roughly the same positions. As long as the change in data is not too large, the gradient descent converges quickly, and the animation looks smooth.

The circles are internally stored in normalized form, with total area equal to 1. Whenever data is returned to the client, the circles are scaled to fit the `[width, height]` window. The normalized circles are still saved internally; they will be needed again if the data is changed. Calling `venn.resize()` does not recompute circle locations with the gradient descent

algorithm; it merely scales the circles.

To compute the SVD, I used the Numeric Javascript library [3].

Wilkenson also describes a clever way to calculate areas of overlap. Calculating the overlap area mathematically is very difficult when the number of circles increases. Wilkenson approximates this calculation by "drawing" each circle on an 200x200 pixel grid, with each circle on a different layer. Elements in a 200x200 matrix are set to 1 if the pixel is in a circle, 0 otherwise. The stacked pixels are counted to see which circles overlay which pixel. Running time for calculating areas of overlap thus increases linearly. While it would be unreasonable to use this type of visualization for too many circles due to goodness-of-fit issues, running time would be reasonable.

Wilkenson's algorithm uses a more time-consuming method to perform a final set of iterations after the gradient descent. I realized that the final set of iterations more than doubled runtime. Results were not noticeably improved. Since D3 clients need good performance for animations and interaction, I removed the final set of iterations.

The stress measure is the sum of squared residuals divided by the total sum of squares. Each residual is simply the difference between a region's desired area (from data) and its actual area (from the current iteration of gradient descent). The stress measure is exposed to the client via function call `venn.stress`, so that the client may decide if the goodness of fit is unacceptable.

It is difficult to recommend a threshold for stress. I would point out that Wilkenson provides an example with correlation 0.99 (stress = 0.02), and yet this diagram suggests that Dorian Gray has no words in common with either Macbeth or the Bible (Figure 8).

RESULTS

Stress

The stress of the diagram in Figure 5 is 0.05. I intended to create a realistic-looking visualization of a user's social network.

The stress of the diagram in Figure 6 is 0.006, which is exceptionally low because the data was hand-tuned. The stress of the diagram in Figure 7 is 0.02.

Running Time

The social network visualization (Figure 5) was generated about 370 milliseconds on a 2010 MacBook Air, including rendering. The diagram in Figure 6 required about 440 milliseconds. The difference occurs primarily because the latter contains one more circle.

The majority of that time is spent in d3.layout.venn(), which calculates the circle positions and sizes. Only 15-20 milliseconds are needed for rendering. When recomputing circles for Figure 7, which follows Figure 6 in a transition, runtime is reduced to roughly two-thirds of the first computation's runtime. Fewer gradient descent iterations are required, and the starting position does not need to be calculated.

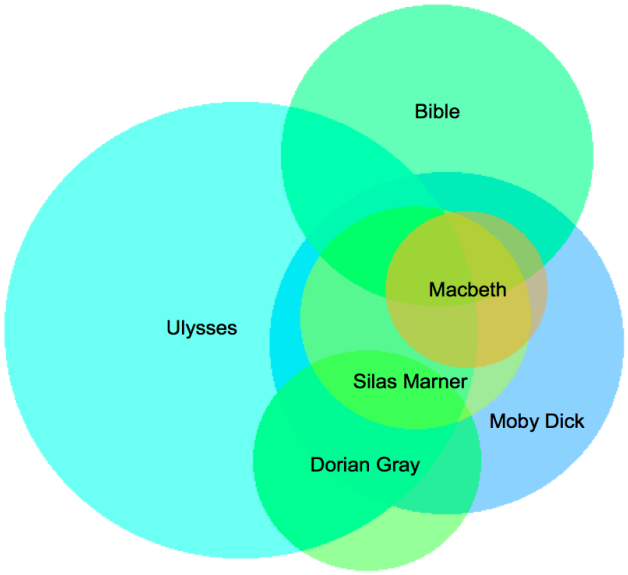

Figure 8. Area-proportional Euler diagram with stress 0.02.

DISCUSSION

The goal of this project was to bring area-proportional Venn diagrams within reach for D3 programmers. An experienced D3 programmer should now be able to create this type of visualization as easily as a pie chart. The syntax is the same as the syntax for any other visualization layout in D3. Inexperienced D3 programmers will have more of a challenge, but most of the difficulty lies with D3 itself. In particular, the code for the animation was not easy to write correctly, because both the circle centers and the circle radii must be updated. With availability of these examples, future users will have an easier task.

Area-proportional Venn diagrams are not commonly seen, although perhaps this library and Wilkenson's recently introduced venneuler library will increase their popularity. Previously, they were very difficult to compute for more than two sets, and especially for more than three sets.

However, it is clear that this visualization type must be used carefully, and not applied indiscriminately. The difficulty lies in that the diagrams usually cannot be perfectly accurate. They are sometimes wildly inaccurate, depending on the dataset.

Venn diagrams have broad appeal to non-technical audiences, because they are so easy to comprehend at a glance. For more technical audiences, they are useful for understanding the data on a high level, but do not show much detail. With more interaction features, though, a visualization designer could build a data exploration tool featuring area-proportional Venn diagrams. If the goal is serious data ex-

ploration, the designer and the user must be careful about high-stress Venn diagrams, because they can mislead.

FUTURE WORK

This implementation of area-proportional Venn diagrams is functional, but the following refinements would be desirable and relatively easy to accomplish:

- Make this feature available as part of D3.
- Offer the option to maintain relative sizes during a transition. Currently, the visualization is scaled to fit the specified width/height. A circle may have no actual change in size during a transition, but it may appear to grow or shrink, making room for other circles. This behavior has its benefits; if all circles' actual data doubles in size, we do not wish to have the entire visualization double in size. However, in certain circumstances, this behavior may be undesirable.
- Provide smarter suggestions for label locations.
- Provide alternate ways to input data.
- Rotate the result such that it better fits a rectangular space. (Currently, we may end up with a tall visualization in a wide box.)
- Better error handling. Currently, there is no error handling when the client enters data in an improper format. However, this seems fairly consistent with D3 in general.
- Allow the client to add or remove circles in a transition application. Currently, the client cannot add circles, but may "remove" circles by setting size to zero. There is an assumption that the number of regions stays constant.
- Allow the client to query for the segment that contains a particular coordinate. This might be useful if the client were to implement mouseovers, for example.

These extensions require more work to achieve:

- Favor the accuracy of some intersections over other intersections. This would require substantial revision to the gradient descent algorithm.
- Support non-circular shapes. This would require a different algorithm.

REFERENCES

1. M. Bostock. Data-driven documents. `http://d3js.org/`.
2. T. Hulsen, J. de Vlieg, and W. Alkema. Biovenn - a web application for the comparison and visualization of biological lists using area-proportional venn diagrams. *BMC Genomics*, 2008. `http://www.cmbi.ru.nl/cdd/biovenn/`.
3. S. Loisel. Numeric javascript. `http://www.numericjs.com/`.
4. L. Micallef and P. Rodgers. eulerape. `http://www.eulerdiagrams.org/eulerAPE/`.
5. L. Wilkinson. venneuler: Venn and euler diagrams. `http://www.rforge.net/venneuler/`.
6. L. Wilkinson. Exact and approximate area-proportional circular venn and euler diagrams. *IEEE Trans. Vis. Comput. Graph.*, 18(2):321–331, 2012.