

## **MODULE II**

Mapping – Time series – Connections and correlations – Area chart – Pivot table – Scatter charts, Scatter maps – Tree maps, Space filling and non-space filling methods – Hierarchies and Recursion – Networks and Graphs – Displaying in Geo Graphs – link graph – Matrix representation for graphs – Info graphics Measures.

# Mapping Concepts in Tableau

If you want to analyze your data geographically, you can plot your data on a map in Tableau. This topic explains why and when you should put your data on a map visualization. It also describes some of the types of maps you can create in Tableau, with links to topics that demonstrate how to create each one.

## Why put your data on a map?

There are many reasons to put your data on a map. Perhaps you have some location data in your data source? Or maybe you think a map could really make your data pop? Both of those are good enough reasons to create a map visualization, but it's important to keep in mind that maps, like any other type of visualization, serve a particular purpose: they answer spatial questions.

You make a map in Tableau because you have a spatial question, and you need to use a map to understand the trends or patterns in your data.

But what is a spatial question? Some examples might be:

- Which state has the most farmers markets?
- Where are the regions in the U.S. with the high obesity rates?
- Which metro station is the busiest for each metro line in my city?
- Where did the storms move over time?
- Where are people checking out and returning bikes from their local bike share program?

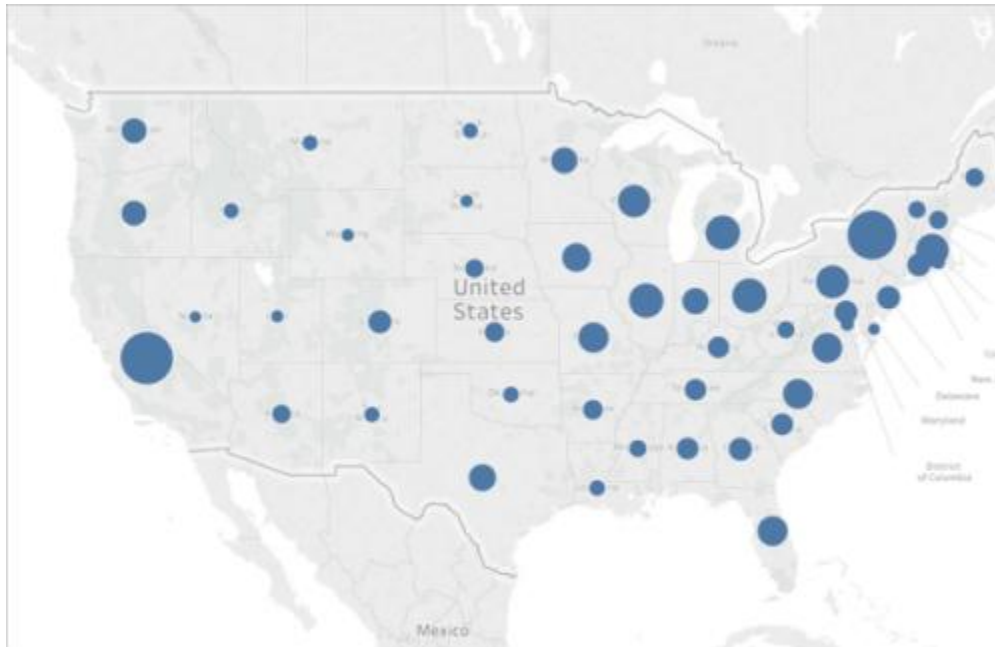
All of these are spatial questions. However, is a map the best way to answer them?

## When should you use a map to represent your data?

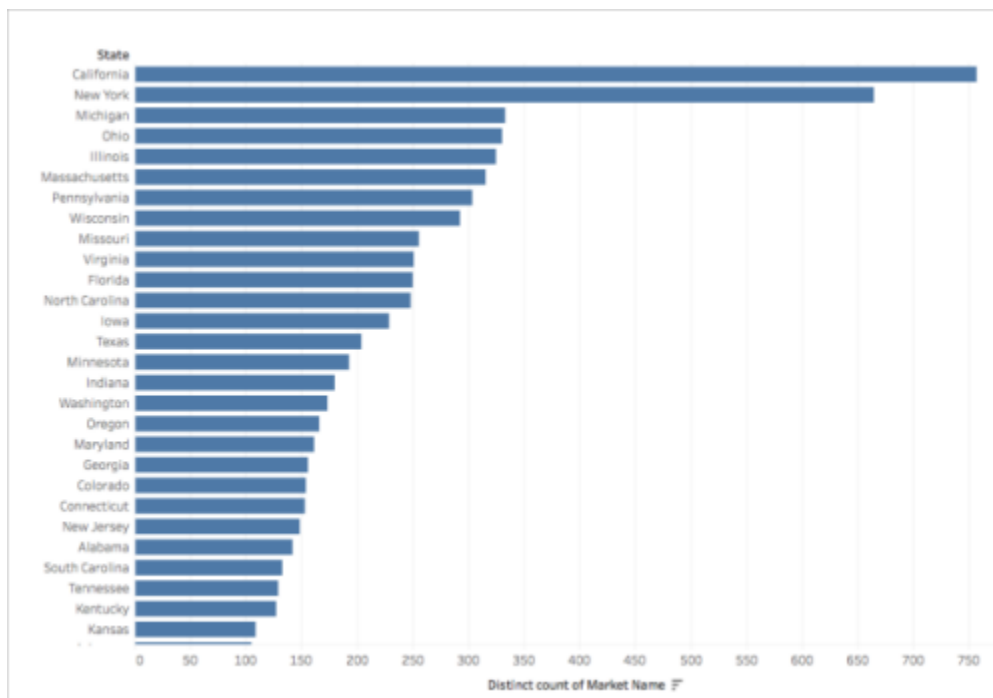
If you have a spatial question, a map view might be a great way to answer it. However, that might not always be the case.

Take for example, the first question from the list above: Which state has the most farmers markets?

If you had a data source with a list of farmers markets per state, you might create a map view like the one below. Can you easily tell the difference between New York and California? Which one has more farmers markets?



What if you create a bar chart instead? Now is it easy to spot the state with the most farmers markets?



The above example is one of many where a different type of visualization would be better to answer a spatial question than a map.

So when do you know if you should use a map view?

One rule of thumb is to ask yourself whether or not you could answer your question faster, or easier with another visualization. If the answer is yes, then perhaps a map view is not the best visualization for the data you're using. If the answer is no, then take the following into account:

Maps that answer questions well have both appropriate data representation, and attractive data representation. In other words: the data is not misleading, and the map is appealing.

If your map is beautiful, but the data is misleading, or not very insightful, you run the risk of people misinterpreting your data. That's why it's important to create maps that represent your data accurately, as well as attractively.

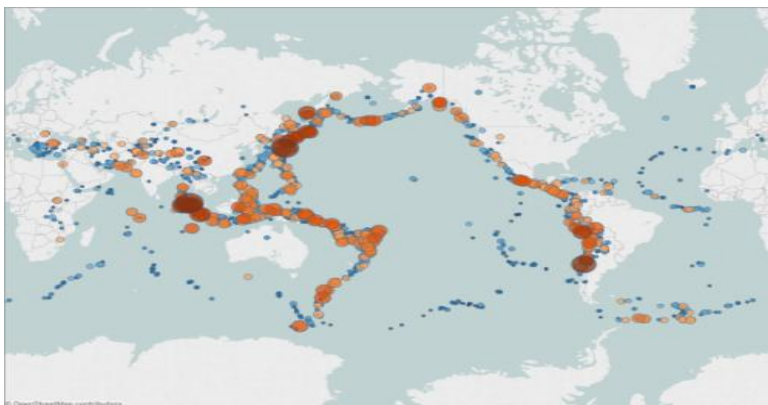
What types of maps can you build in Tableau?

With Tableau, you can create the following common map types:

- [Proportional symbol maps](#)
- [Choropleth maps \(filled maps\)](#)
- [Point distribution maps](#)
- [Flow maps \(path maps\)](#)

### **Proportional symbol maps**

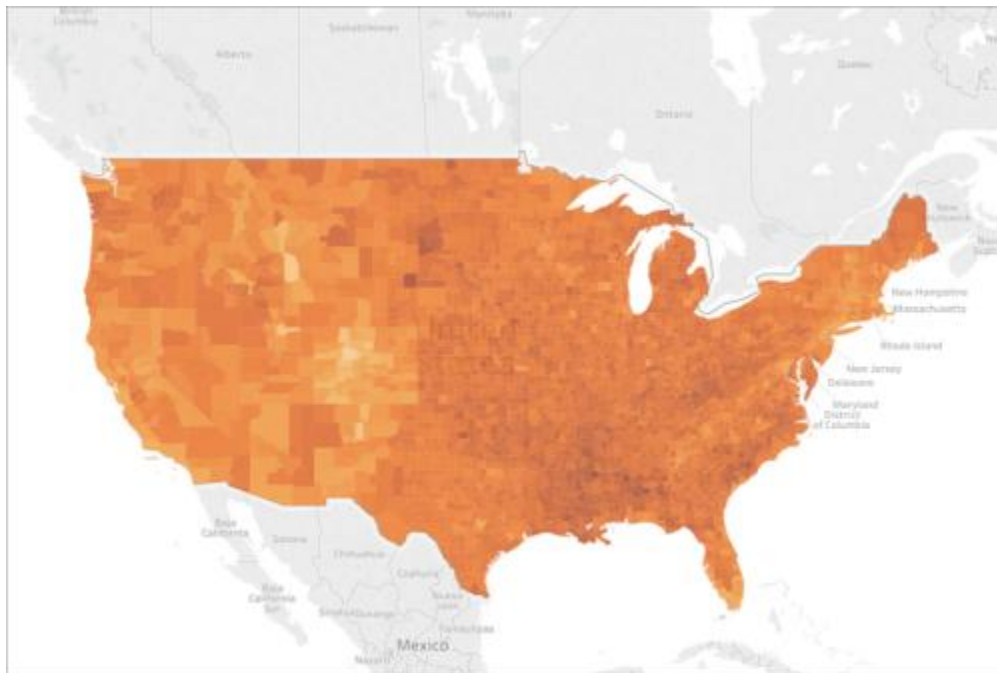
Proportional symbol maps are great for showing quantitative data for individual locations. For example, you can plot earthquakes around the world and size them by magnitude.



### **Choropleth maps (filled maps)**

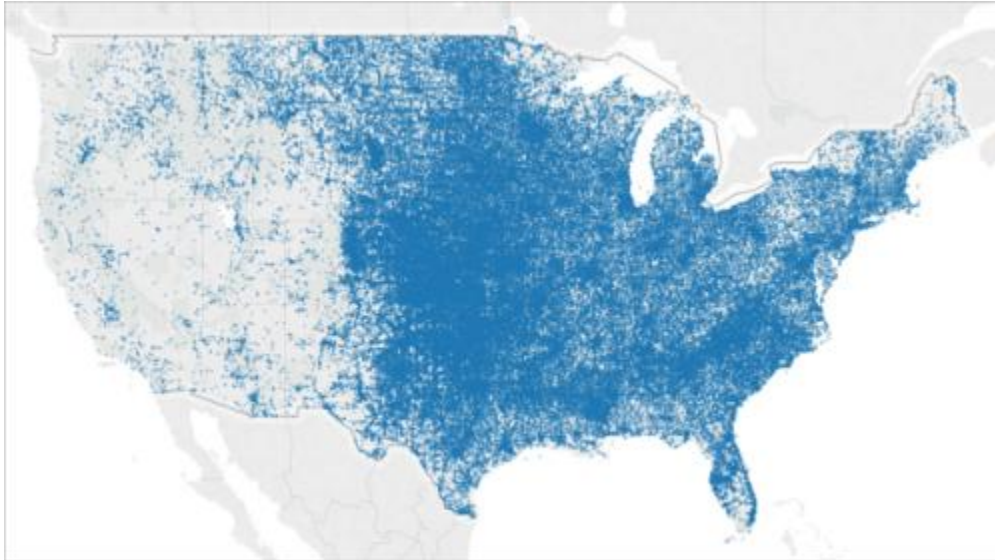
Also known as filled maps in Tableau, Choropleth maps are great for showing ratio data. For example, if you want to see obesity rates for every county across the United States, you might consider creating a choropleth map to see if you can spot any spatial trends.

For more information about Choropleth maps, and to learn how to create them in Tableau, see [Create Maps that Show Ratio or Aggregated Data in Tableau](#)(Link opens in a new window).



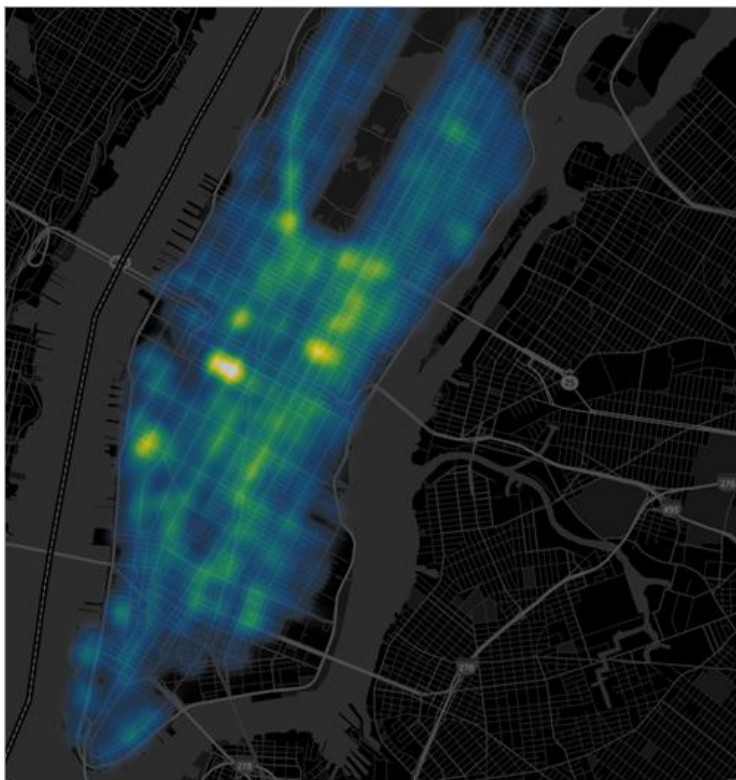
### **Point distribution maps**

Point distribution maps can be used when you want to show approximate locations and are looking for visual clusters of data. For example, if you want to see where all the hailstorms were in the U.S. last year, you can create a point distribution map to see if you can spot any clusters.



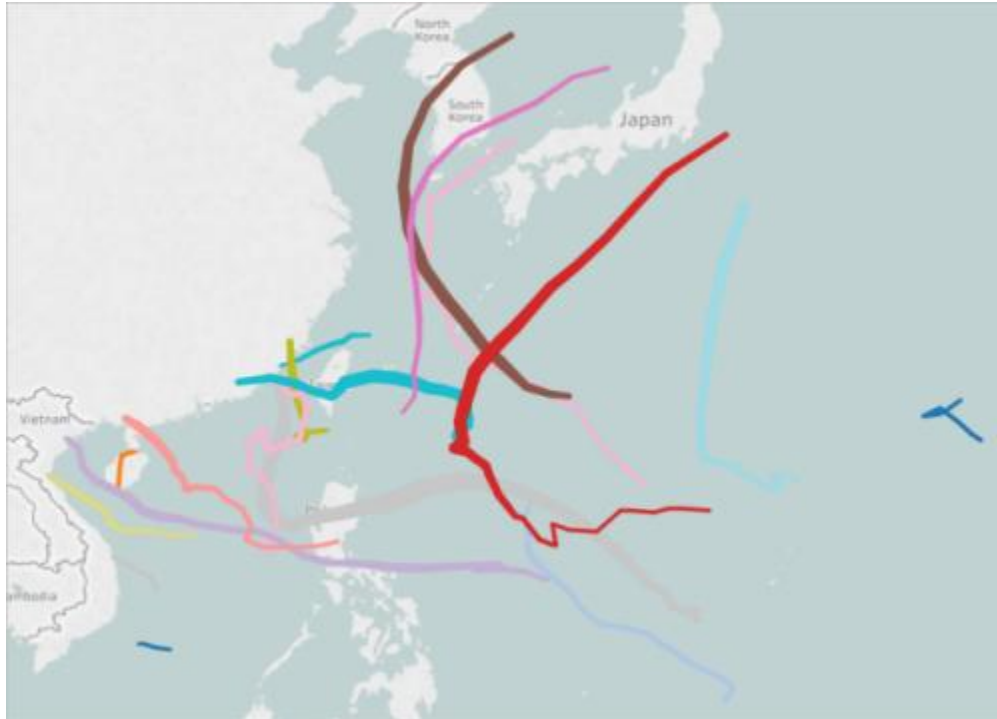
### density maps

density maps, can be used when you want to show a trend for visual clusters of data. For example, if you want to find out which areas of Manhattan have the most taxi pickups, you can create a density map to see which areas are most popular.



### **Flow maps (path maps)**

You can use flow maps to connect paths across a map and to see where something went over time. For example, you can track the paths of major storms across the world over a period of time.



# Mapping.- using Processing tool

This chapter covers the basics of reading, displaying, and interacting with a data set. As an example, we'll use a map of the United States, and a set of data values for all 50 states. Drawing such a map is a simple enough task that could be done without programming—either with mapping software or by hand—but it gives us an example upon which to build.

## Drawing a Map

Some development environments separate work into projects; the equivalent term for Processing is a *sketch*. Start a new Processing sketch by selecting File → New.

For this example, we'll use a map of the United States to use as a background image. The map can be downloaded from <http://benfry.com/writing/map/map.png>.

Drag and drop the *map.png* file into the Processing editor window. A message at the bottom will appear confirming that the file has been added to the sketch. You can also add files by selecting Sketch → Add File. A sketch is organized as a folder, and all data files are placed in a subfolder named *data*. (The *data* folder is covered in Chapter 2.)

Then, enter the following code:

```
PImage mapImage;

void setup() {
  size(640, 400);
  mapImage = loadImage("map.png");
}
```

```
void draw() {
  background(255);
  image(mapImage, 0, 0);
}
```

Finally, click the Run button. Assuming everything was entered correctly, a map of the United States will appear in a new window.



## Explanation of the Processing Code

Processing API functions are named to make their uses as obvious as possible. Method names, such as `loadImage()`, convey the purpose of the calls in simple language. What you may need to get used to is dividing your code into functions such as `setup()` and `draw()`, which determine how the code is handled. After clicking the Run button, the `setup()` method executes once. After `setup()` has completed, the `draw()` method runs repeatedly. Use the `setup()` method to load images, fonts, and set initial values for variables. The `draw()` method runs at 60 frames per second (or slower if it takes longer than 1/60th of a second to run the code inside the `draw()` method); it can be used to update the screen to show animation or respond to mouse movement and other types of input.

Our first function calls are very basic. The `loadImage()` function reads an image from the data folder (URLs or absolute paths also work). The `PImage` class is a container for image data, and the `image()` command draws it to the screen at a specific location.

In the previous section, we saw how displaying a map in Processing is a two-step process:

1. Load the data.
2. Display the data in the desired format.

Displaying the centers of states follows the same pattern, although a little more code is involved:

1. Create `locationTable` and use the `locationTable.getFloat()` function to read each location's coordinates (x and y values).
2. Draw a circle using those values. Because a circle, geometrically speaking, is just an ellipse whose width and height are the same, graphics libraries provide an ellipse-drawing function that covers circle drawing as well.

The `smooth()`, `fill()`, and `noStroke()` functions apply to any drawing we subsequently do in the `draw()` function.

---

A new version of the code follows, with modifications highlighted:

```
PImage mapImage;
Table locationTable;
int rowCount;

void setup() {
  size(640, 400);
  mapImage = loadImage("map.png");
  // Make a data table from a file that contains
  // the coordinates of each state.
  locationTable = new Table("locations.tsv");
  // The row count will be used a lot, so store it globally.
  rowCount = locationTable.getRowCount();
}

void draw() {
  background(255);
  image(mapImage, 0, 0);

  // Drawing attributes for the ellipses.
  smooth();
  fill(192, 0, 0);
  noStroke();

  // Loop through the rows of the locations file and draw the points.
  for (int row = 0; row < rowCount; row++) {
    float x = locationTable.getFloat(row, 1); // column 1
    float y = locationTable.getFloat(row, 2); // column 2
    ellipse(x, y, 9, 9);
  }
}
```

The figure shows the map and points for each location.



## Data on a Map

Next we want to load a set of values that will appear on the map itself. For this, we add another `Table` object and load the data from a file called *random.tsv*, available at <http://benfry.com/writing/map/random.tsv>.

It's always important to find the minimum and maximum values for the data, because that range will need to be mapped to other features (such as size or color) for display. To do this, use a `for` loop to walk through each line of the data table and check to see whether each value is bigger than the maximum found so far, or smaller than the minimum. To begin, the `dataMin` variable is set to `MAX_FLOAT`, a built-in value for the maximum possible float value. This ensures that `dataMin` will be replaced with the first value found in the table. The same is done for `dataMax`, by setting it to `MIN_FLOAT`. Using 0 instead of `MIN_FLOAT` and `MAX_FLOAT` will not work in cases where the minimum value in the data set is a positive number (e.g., 2.4) or the maximum is a negative number (e.g., -3.75).

The data table is loaded in the same fashion as the location data, and the code to find the minimum and maximum immediately follows:

```
PImage mapImage;  
Table locationTable;  
int rowCount;
```

```
Table dataTable;
```

```
float dataMin = MAX_FLOAT;
```

```
float dataMax = MIN_FLOAT;
```

```
void setup( ) {
```

```
  size(640, 400);
```

```
  mapImage = loadImage("map.png");
```

```
  locationTable = new Table("locations.tsv");
```

```
  rowCount = locationTable.getRowCount( );
```

```
  // Read the data table.
```

```
  dataTable = new Table("random.tsv");
```

```
  // Find the minimum and maximum values.
```

```
  for (int row = 0; row < rowCount; row++) {
```

```
    float value = dataTable.getFloat(row, 1);
```

```
    if (value > dataMax) {
```

```
      dataMax = value;
```

```
    }
```

```

if (value < dataMin) {
dataMin = value;
}
}
}

```

The other half of the program (shown later) draws a data point for each location. A `drawData()` function is introduced, which takes `x` and `y` coordinates as parameters, along with an abbreviation for a state. The `drawData()` function grabs the float value from column 1 based on a state abbreviation (which can be found in column 0).

The `getRowName()` function gets the name of a particular row. This is just a convenience function because the row name is usually in column 0, so it's identical to `getString(row, 0)`. The row titles for this data set are the two-letter state abbreviations. In the modified example, `getRowName()` is used to get the state abbreviation for each row of the data file.

The `getFloat()` function can also use a row name instead of a row number, which simply matches the `String` supplied against the abbreviation found in column 0 of the *random.tsv* data file. The results are shown in Figure 3-2.

The rest of the program follows:

```

void draw() {
  background(255);
  image(mapImage, 0, 0);

  smooth();
  fill(192, 0, 0);
  noStroke();

```

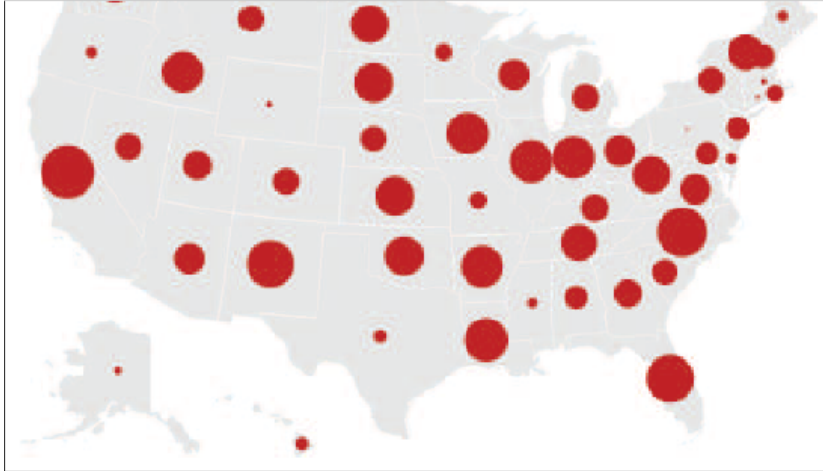


Figure 3-2. Varying data by size

```

for (int row = 0; row < rowCount; row++) {
    String abbrev = dataTable.getRowName(row);
    float x = locationTable.getFloat(abbrev, 1);
    float y = locationTable.getFloat(abbrev, 2);
    drawData(x, y, abbrev);
}

// Map the size of the ellipse to the data value
void drawData(float x, float y, String abbrev) {
    // Get data value for state
    float value = dataTable.getFloat(abbrev, 1);
    // Re-map the value to a number between 2 and 40
    float mapped = map(value, dataMin, dataMax, 2, 40);
    // Draw an ellipse for this item
    ellipse(x, y, mapped, mapped);
}

```

The `map()` function converts numbers from one range to another. In this case, value is expected to be somewhere between `dataMin` and `dataMax`. Using `map()` re-proportions value to be a number between 2 and 40. The `map()` function is useful for hiding the math involved in the conversion, which makes code quicker to write and easier to read. A lot of visualization problems revolve around mapping data from one range to another.

The `lerp()` function converts a normalized value to another range (`norm()` and `lerp()` together make up the `map()` function), and the `lerpColor()` function does the same except it interpolates between two colors. The syntax:

---

`color between = lerpColor(color1, color2, percent)`

---

# TIME SERIES

Time series data is data that is collected at different points in time. This is opposed to cross-sectional data which observes individuals, companies, etc. at a single point in time.

Because data points in time series are collected at adjacent time periods there is potential for correlation between observations. This is one of the features that distinguishes time series data from cross-sectional data.

Time series data can be found in [economics](#), [social sciences](#), [finance](#), [epidemiology](#), and the [physical sciences](#).

## What Is an Example of Time Series Data?

Field	Example topics	Example dataset
Economics	Gross Domestic Product (GDP), Consumer Price Index (CPI), S&P 500 Index, and unemployment rates	<a href="#">U.S. GDP from the Federal Reserve Economic Data</a>
Social sciences	Birth rates, population, migration data, political indicators	<a href="#">Population without citizenship from Eurostat</a>
Epidemiology	Disease rates, mortality rates, mosquito populations	<a href="#">U.S. Cancer Incidence rates from the Center for Disease Control</a>
Medicine	Blood pressure tracking, weight tracking, cholesterol measurements, heart rate monitoring	<a href="#">MRI scanning and behavioral test dataset</a>

## What Is an Example of Time Series Data?

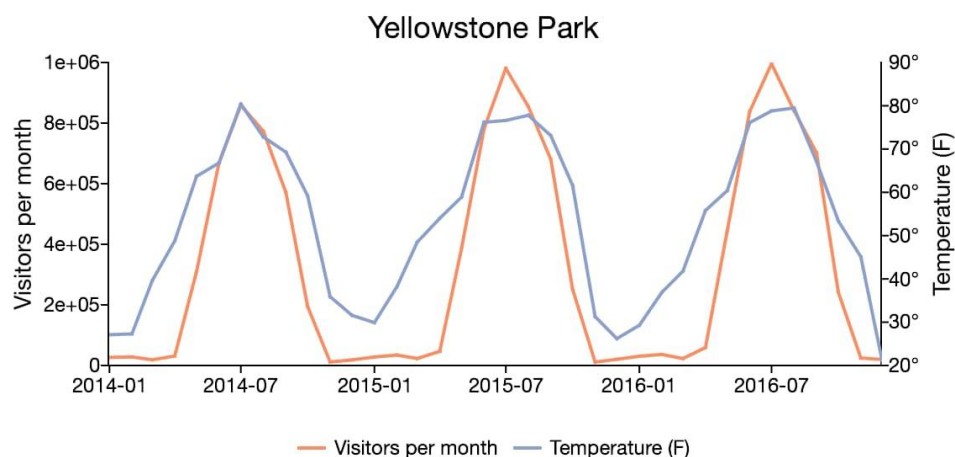
Field	Example topics	Example dataset
Physical sciences	Global temperatures, monthly sunspot observations, pollution levels.	<a href="#">Global air pollution from the Our World in Data</a>

The statistical characteristics of time series data often violate the assumptions of conventional statistical methods. Because of this, analyzing time series data requires a unique set of tools and methods, collectively known as time series analysis.

This article covers the fundamental concepts of time series analysis and should give you a foundation for working with time series data.

## What Is Time Series Data?

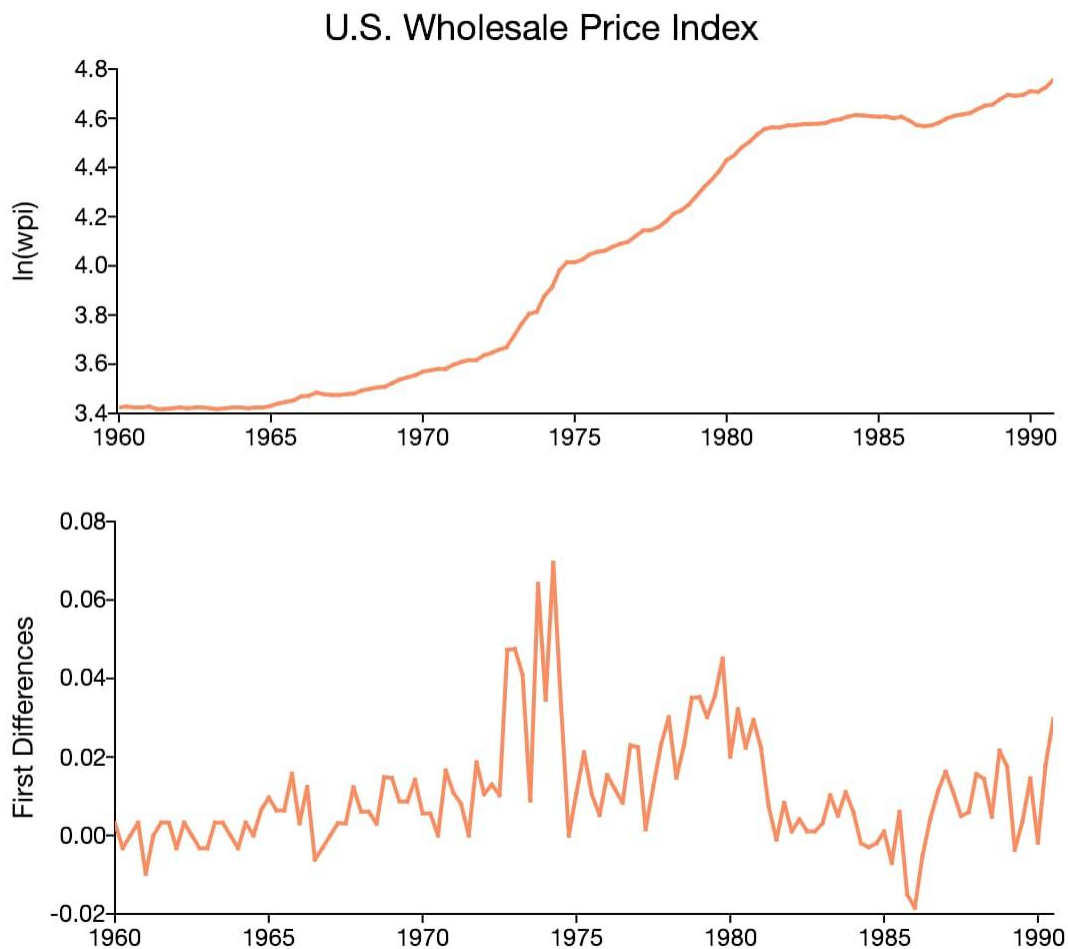
Time series data is a collection of quantities that are assembled over even intervals in time and ordered chronologically. The time interval at which data is collection is generally referred to as the time series frequency.



For example, the [time series graph](#) above plots the visitors per month to Yellowstone National Park with the average monthly temperatures. The data ranges between January 2014 to December 2016 and is collected at a monthly frequency.

## Time Series Visualization

---



### What is a time series graph?

A [time series graph](#) plots observed values on the y-axis against an increment of time on the x-axis. These graphs visually highlight the behavior and patterns of the data and can lay the foundation for building a reliable model.



More specifically, [visualizing time series data](#) provides a preliminary tool for detecting if data:

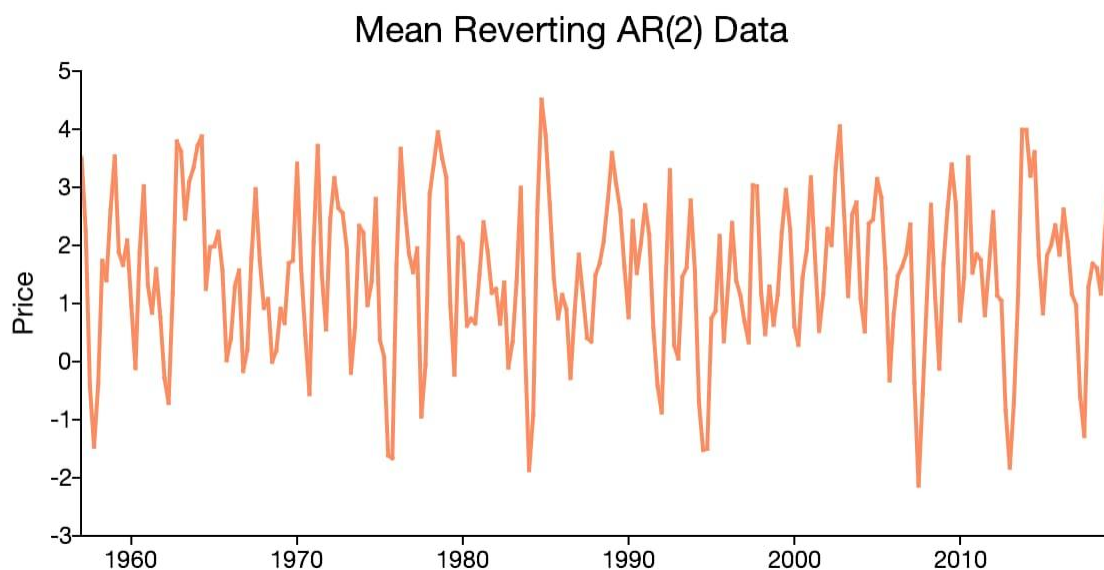
- Is mean-reverting or has explosive behavior;
- Has a time trend;
- Exhibits seasonality;
- Demonstrates structural breaks.

This, in turn, can help guide the testing, diagnostics, and estimation methods used during time series modeling and analysis.

## Mean Reverting Data

Mean reverting data returns, over time, to a time-invariant mean. It is important to know whether a model includes a non-zero mean because it is a prerequisite for determining appropriate testing and modeling methods.

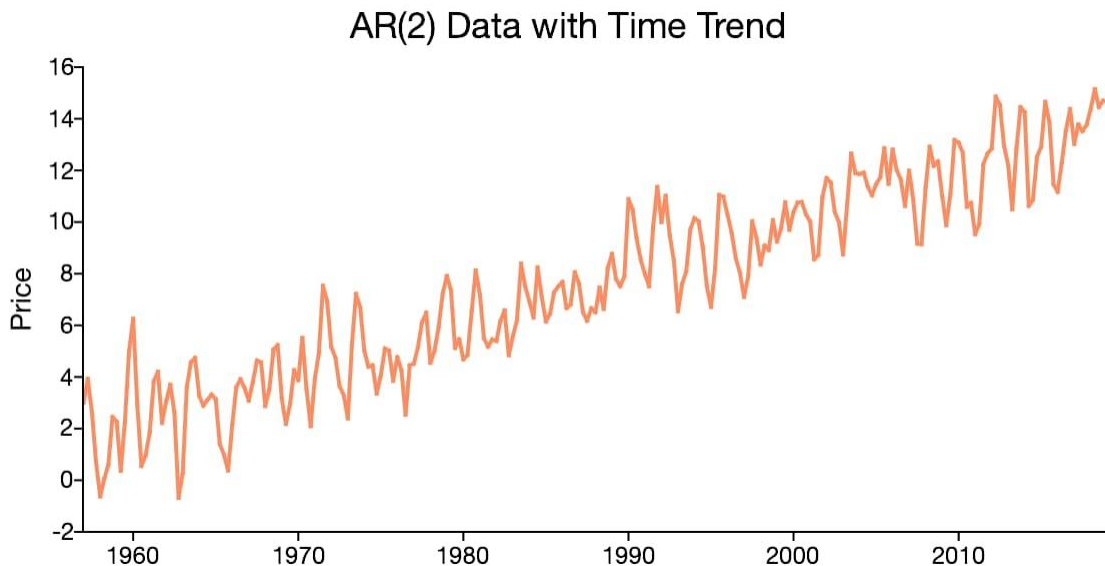
For example, unit root tests use different regressions, statistics, and distributions when a non-zero constant is included in the model.



A [time series graph](#) provides a tool for visually inspecting if the data is mean-reverting, and if it is, what mean the data is centered around. While visual inspection should never replace statistical estimation, it can help you decide whether a non-zero mean should be included in the model.

For example, the data in the figure above varies around a mean that lies above the zero line. This indicates that the models and tests for this data must incorporate a non-zero mean.

## Time Trending Data



In addition to containing a non-zero mean, time series data may also have a deterministic component that is proportionate to the time period. When this occurs, the time series data is said to have a time trend.

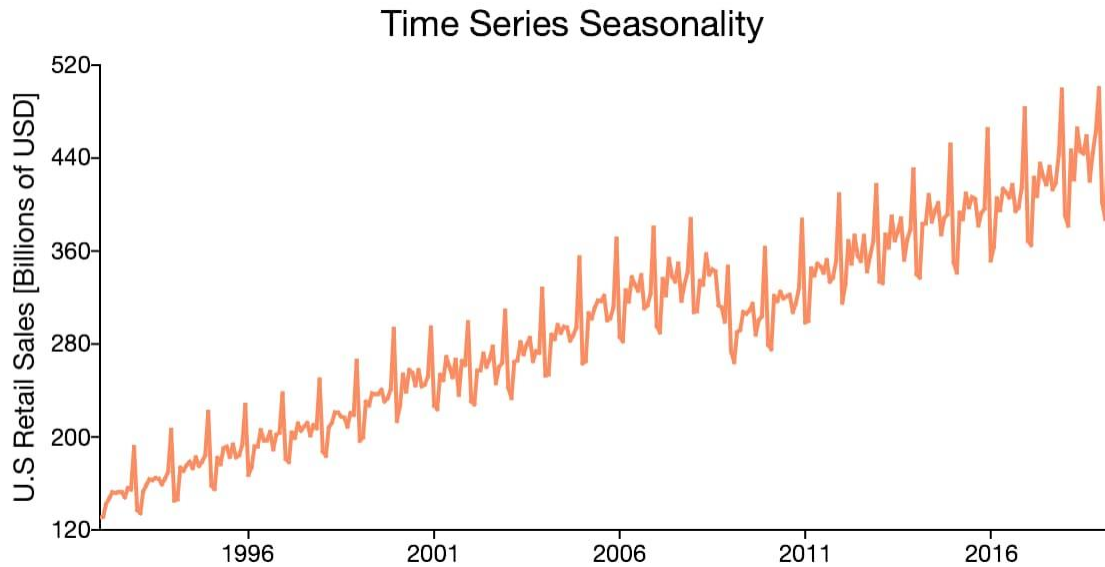
Time trends in time series data also have implications for testing and modeling. The reliability of a time series model depends on properly identifying and accounting for time trends.

A time series plot which looks like it centers around an increasing or decreasing line, like that in the plot above, suggests the presence of a time trend.

## Seasonality

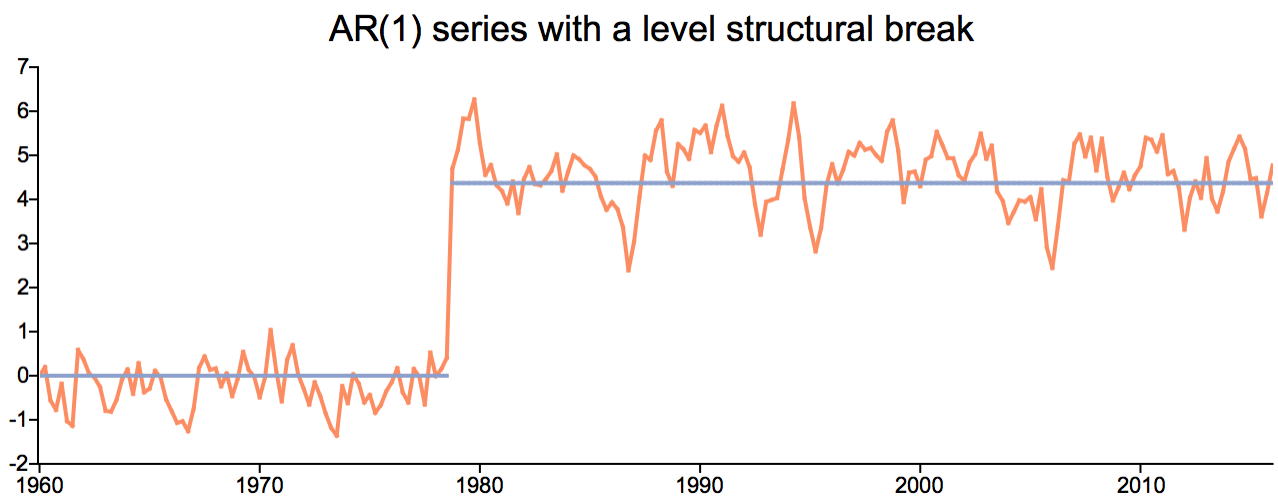
Seasonality is another characteristic of time series data that can be visually identified in [time series plots](#). Seasonality occurs when time series data exhibits regular and predictable patterns at time intervals that are smaller than a year.

An example of a time series with [seasonality](#) is retail sales, which often increase between September to December and will decrease between January and February.



## Structural Breaks

Sometimes time series data shows a sudden change in behavior at a certain point in time. For example, many macroeconomic indicators changed sharply in 2008 after the start of the global financial crisis. These sudden changes are often referred to as [structural breaks](#) or non-linearities.



These [structural breaks](#) can create instability in the parameters of a model. This, in turn, can diminish the validity and reliability of that model.

Models of time series analysis include:

- **Classification:** Identifies and assigns categories to the data.
- **Curve fitting:** Plots the data along a curve to study the relationships of variables within the data.
- **Descriptive analysis:** Identifies patterns in time series data, like trends, cycles, or seasonal variation.
- **Explanative analysis:** Attempts to understand the data and the relationships within it, as well as cause and effect.
- **Exploratory analysis:** Highlights the main characteristics of the time series data, usually in a visual format.
- **Forecasting:** Predicts future data. This type is based on historical trends. It uses the historical data as a model for future data, predicting scenarios that could happen along future plot points.
- **Intervention analysis:** Studies how an event can change the data.
- **Segmentation:** Splits the data into segments to show the underlying properties of the source information.

## Data classification

Further, time series data can be classified into two main categories:

- **Stock time series data** means measuring attributes at a certain point in time, like a static snapshot of the information as it was.
- **Flow time series data** means measuring the activity of the attributes over a certain period, which is generally part of the total whole and makes up a portion of the results.

Here are some of the [most common types of time series visualizations](#).

### Line graph

Line graph is probably the most simple way to illustrate time series data. It uses points connected to visualize the changes. Being the independent variable, time in line graphs is always presented as the horizontal axis.

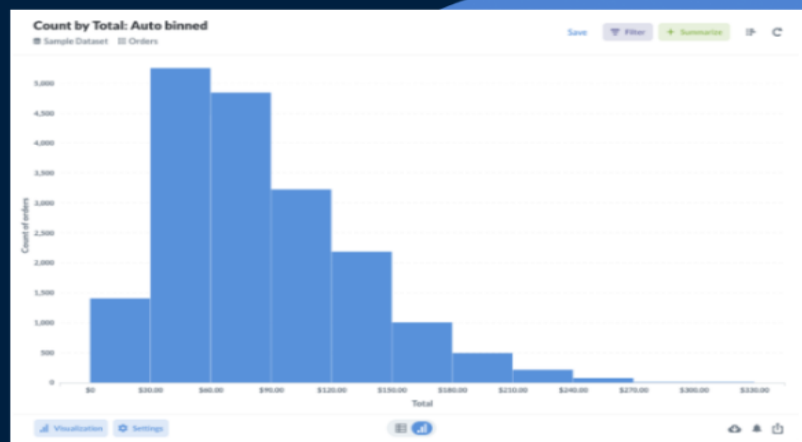
## Line Graph



## Histogram charts

Histogram charts visualize the data by grouping it into bins, where bins are displayed as segmented columns. All bins in a histogram have equal width, while their height is proportional to the number of data points in the bin.

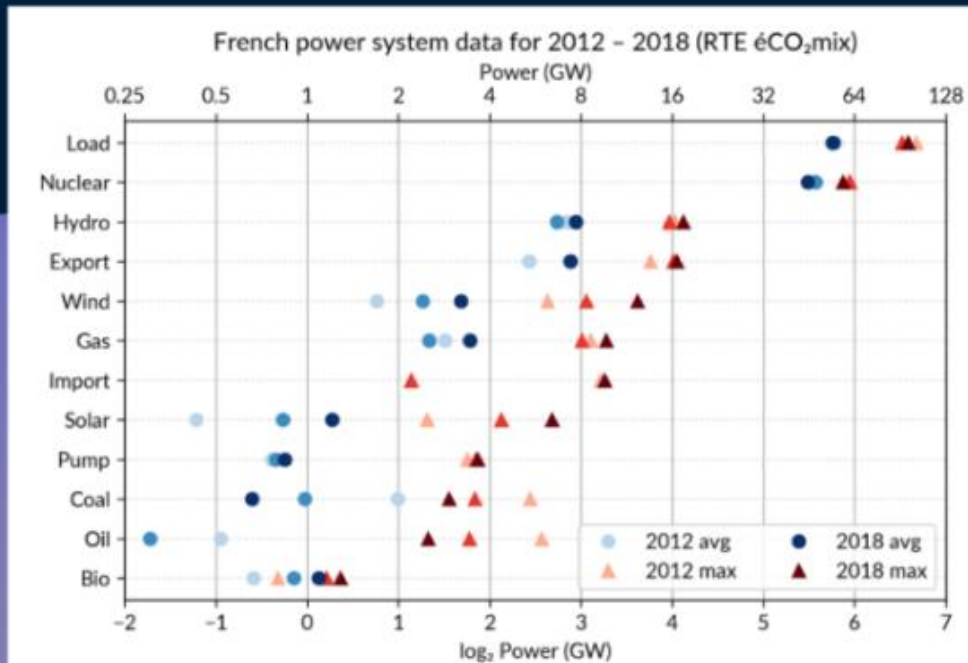
## Histogram Charts



## Dot plots

Dot plots or dot graphs present data points vertically with dot-like markers, with the height of each marker group representing the frequency of the elements in each interval.

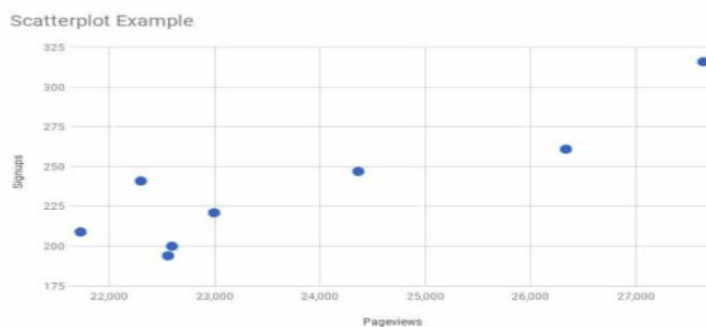
### Dot Plots



## Scatter plots

Scatter plots or charts use the same dot-like markers that are scattered across the chart area of the plot, representing each data point. Scatter plots are usually used as a way to visualize random variables.

### Scatter Plots



## *Trend line*

Trend line is based on standard line and plot graphs, adding a straight line that has to connect at least two points on a chart, extending forward into the future to identify areas of support and resistance.



# Time Series

The time series is a ubiquitous type of data set. It describes how some measurable feature (for instance, population, snowfall, or items sold) has changed over a period of time. Edward Tufte credits Johann Heinrich Lambert with the formal introduction of the time series to scientific literature in the 1700s.\*

Because of its ubiquity, the time series is a good place to start when learning about visualization. With it we can cover:

- Acquiring a table of data from a text file
- Parsing the contents of the file into a usable data structure
- Calculating the boundaries of the data to facilitate representation
- Finding a suitable representation and considering alternatives
- Refining the representation with consideration for placement, type, line weight, and color
- Providing a means of interacting with the data so that we can compare variables against one another or against the average of the whole data set



## Milk, Tea, and Coffee (Acquire and Parse)

The data set we use was originally downloaded from <http://www.ers.usda.gov/Data/FoodConsumption/FoodAvailQueryable.aspx>.

The page lets you define a query to download a data set of interest. The site claims that the data is in Excel format, but a glance at the contents of the resulting file shows that it's only an HTML file with an *.xls* extension that fools Excel into opening it. Rather than getting into the specifics of how to download and clean the data, I offer an already processed version here:

<http://benfry.com/writing/series/milk-tea-coffee.tsv>

This data set contains three columns: the first for milk, the second for coffee, and the third for tea consumption in the United States from 1910 to 2004.

To read this file, use this modified version of the `Table` class from the previous chapter:

<http://benfry.com/writing/series/FloatTable.pde>

The modified version handles data stored as `float` values, making it more efficient than the previous version, which simply converted the data whenever `getString()`, `getFloat()`, or `getInt()` were used.

Open Processing and start a new sketch. Add both files to the sketch by either dragging each into the editor window or using Sketch → Add File.

To begin the representation, it's first necessary to set the boundaries for the plot location. The `plotX1`, `plotY1`, `plotX2`, and `plotY2` variables define the corners of the plot. To provide a nice margin on the left, set `plotX1` to 50, and then set the `plotX2` coordinate by subtracting this value from `width`. This keeps the two sides even, and requires only a single change to adjust the position of both. The same technique is used for the vertical location of the plot:

```
FloatTable data;
float dataMin, dataMax;

float plotX1, plotY1;
float plotX2, plotY2;

int yearMin, yearMax;
int[] years;

void setup() {
  size(720, 405);

  data = new FloatTable("milk-tea-coffee.tsv");

  years = int(data.getRowNames());
  yearMin = years[0];
  yearMax = years[years.length - 1];

  dataMin = 0;
  dataMax = data.getTableMax();

  // Corners of the plotted time series
  plotX1 = 50;
  plotX2 = width - plotX1;
  plotY1 = 60;
  plotY2 = height - plotY1;

  smooth();
}
```

The `rect()` function normally takes the form `rect(x, y, width, height)`, but `rectMode(CORNERS)` changes the parameters to `rect(left, top, right, bottom)`, which is useful because our plot's shape is defined by the corners. Like other methods that affect drawing properties, such as `fill()` and `stroke()`, `rectMode()` affects all geometry that is drawn after it until the next time `rectMode()` is called:

```
void draw() {
    background(224);

    // Show the plot area as a white box.
    fill(255);
    rectMode(CORNERS);
    noStroke();
    rect(plotX1, plotY1, plotX2, plotY2);

    strokeWeight(5);
    // Draw the data for the first column.
    stroke(#5679C1);
    drawDataPoints(0);
}

// Draw the data as a series of points.
void drawDataPoints(int col) {
    int rowCount = data.getRowCount();
    for (int row = 0; row < rowCount; row++) {
        if (data.isValid(row, col)) {
            float value = data.getFloat(row, col);
            float x = map(years[row], yearMin, yearMax, plotX1, plotX2);
            float y = map(value, dataMin, dataMax, plotY2, plotY1);
            point(x, y);
        }
    }
}
```

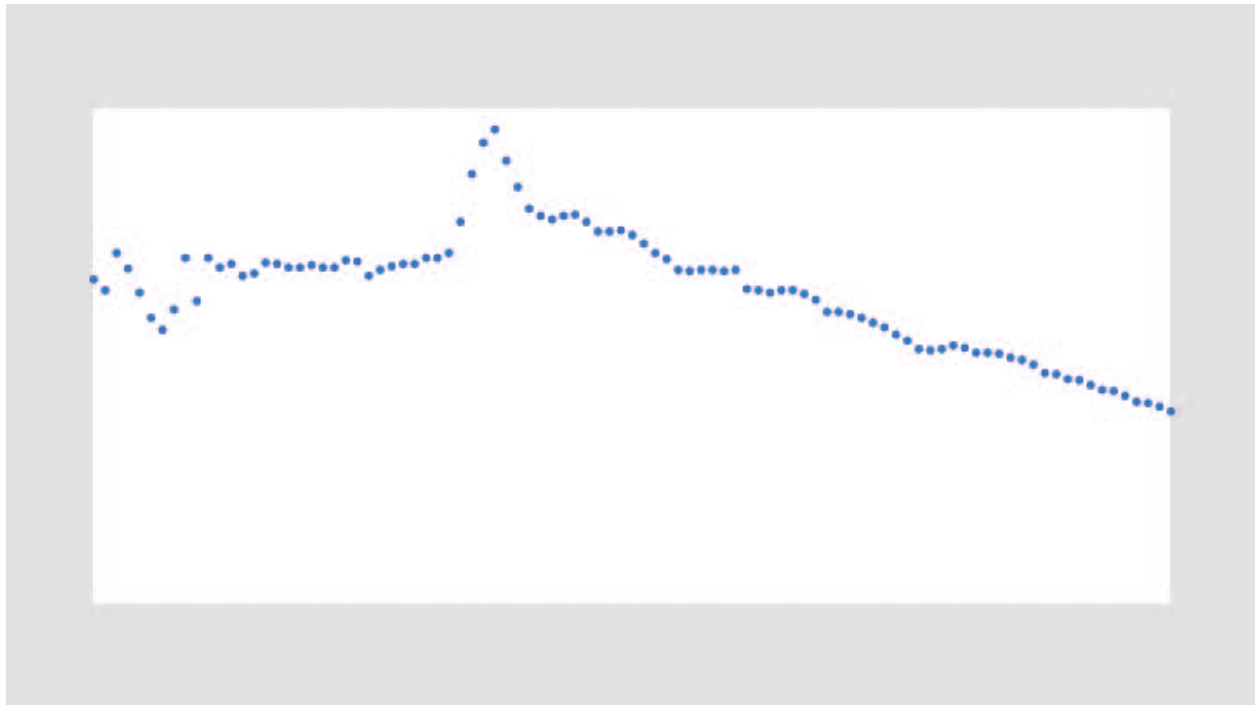
Because the data is drawn as points using the `drawDataPoints()` method, a stroke color and weight are set. This method also takes a column index to draw as a parameter. The results are in Figure 4-1. For the first step, I've shown only the first column of data (the values for milk consumption).

The `map()` function does most of the work. The x coordinate is calculated by mapping the year for each row from `yearMin` and `yearMax` to `plotX1` and `plotX2`. Another option would be to use the `row` variable, instead of the year:

```
float x = map(row, 0, rowCount-1, plotX1, plotX2);
```

But a value for `row` would be less accurate because a year or two might be missing from the data set, which would skew the representation. Again, this data set is complete, but often that is not the case.

Figure 4-1. One set of points over time



Others steps

Labeling the Current Data Set (Refine and Interact)

Drawing Axis Labels (Refine)

## Year Labels

Creating the year axis is straightforward. The data ranges from 1910 to 2004, so an interval of 10 years means marking 10 individual years: 1910, 1920, 1930, and so on, up to 2000. Add the `yearInterval` variable to the beginning of the code before `setup()`:

```
int yearInterval = 10;
```

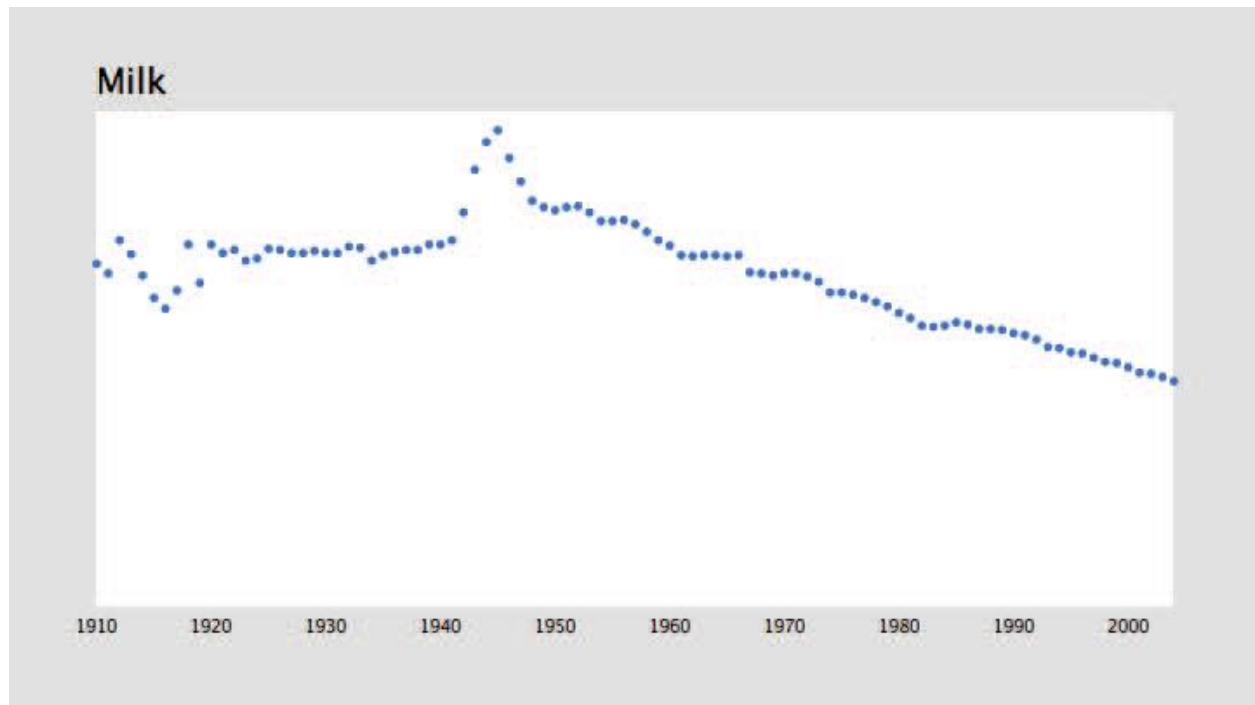
Next, add the following function to draw the year labels:

```
void drawYearLabels() {  
  fill(0);  
  textSize(10);  
  textAlign(CENTER, TOP);  
  for (int row = 0; row < rowCount; row++) {  
    if (years[row] % yearInterval == 0) {  
      float x = map(years[row], yearMin, yearMax, plotX1, plotX2);  
      text(years[row], x, plotY2 + 10);  
    }  
  }  
}
```

The fill color is set to black, the text size to 10, and the alignment to the middle so that the year number centers on the position of the data point for that year.

Two lines in this code deserve further consideration. The first is the line that makes use of the `%`, or *modulo*, operator. A modulo operation returns the remainder from a division. So, for example, `7 % 2` is equal to 1, and `8 % 5` equals 3. It's useful for drawing labels because it provides a way to easily identify a year ending in 0. Dividing 1910 by 10 returns 0, so a label is drawn, whereas dividing 1911 by 10 produces 1, and so it continues until the loop reaches 1920, which also returns 0 when divided by 10.

A second parameter to `textAlign()` sets the vertical alignment of the text. The options are `TOP`, `BOTTOM`, `CENTER`, and `BASELINE` (the default). The `TOP` and `CENTER` parameters are straightforward. The `BOTTOM` parameter is the same as `BASELINE` when only one line of text is used, but for multiple lines, the final line will be aligned to the baseline, with the previous lines appearing above it. When only one parameter is used, the vertical alignment resets to `BASELINE`.



**Figure 4-3.** Time series with labeled x-axis

---

## Connections and Correlations

Data that varies across multiple dimensions is common, and it can be difficult to represent in traditional charts that exploit only the two dimensions of the screen or printed page. In particular, you often have an independent variable and a dependent variable that change over time. Many techniques for representing change exist, but one of the most engaging ways is animation.

In this chapter, we'll create a display of baseball results to explore how relationships can be instantly and powerfully conveyed through the spatial arrangement of data, visual elements such as icons and lines, and most significantly, the use of animation. You don't have to understand baseball to understand this chapter; it's less about the game than it is about the numbers and depicting those numbers.

```

static final int ROW_HEIGHT = 23;
static final float HALF_ROW_HEIGHT = ROW_HEIGHT / 2.0;
static final int SIDE_PADDING = 30;

```

The text size set earlier is about half the height of each row. This creates easy-to-read, double-spaced text. The text itself needn't be particularly large or prominent because it is not as important as the correlation line itself.

The `SIDE_PADDING` variable is used to set a border around the display, adding some whitespace to the edges. The amount should be more than the row height so that it looks intentional, but not so large as to waste space.

The `draw()` method reads as follows:

```

void draw() {
    background(255);
    smooth();

    translate(SIDE_PADDING, SIDE_PADDING);

    float leftX = 160;
    float rightX = 335;

    textAlign(LEFT, CENTER);

    for (int i = 0; i < teamCount; i++) {
        fill(0);
        float standingsY = standings.getRank(i)*ROW_HEIGHT + HALF_ROW_HEIGHT;
        image(logos[i], 0, standingsY - logoHeight/2, logoWidth, logoHeight);
        text(teamNames[i], 28, standingsY);
        text(standings.getTitle(i), 115, standingsY);

        float salaryY = salaries.getRank(i)*ROW_HEIGHT + HALF_ROW_HEIGHT;































        stroke(0);
        line(leftX, standingsY, rightX, salaryY);

        text(salaries.getTitle(i), rightX+10, salaryY);
    }
}

```

The `translate()` method moves the coordinate system over slightly, giving us a white border: (0, 0) will now be (30, 30), so nothing will be drawn in the left or top 30 pixels of the image.



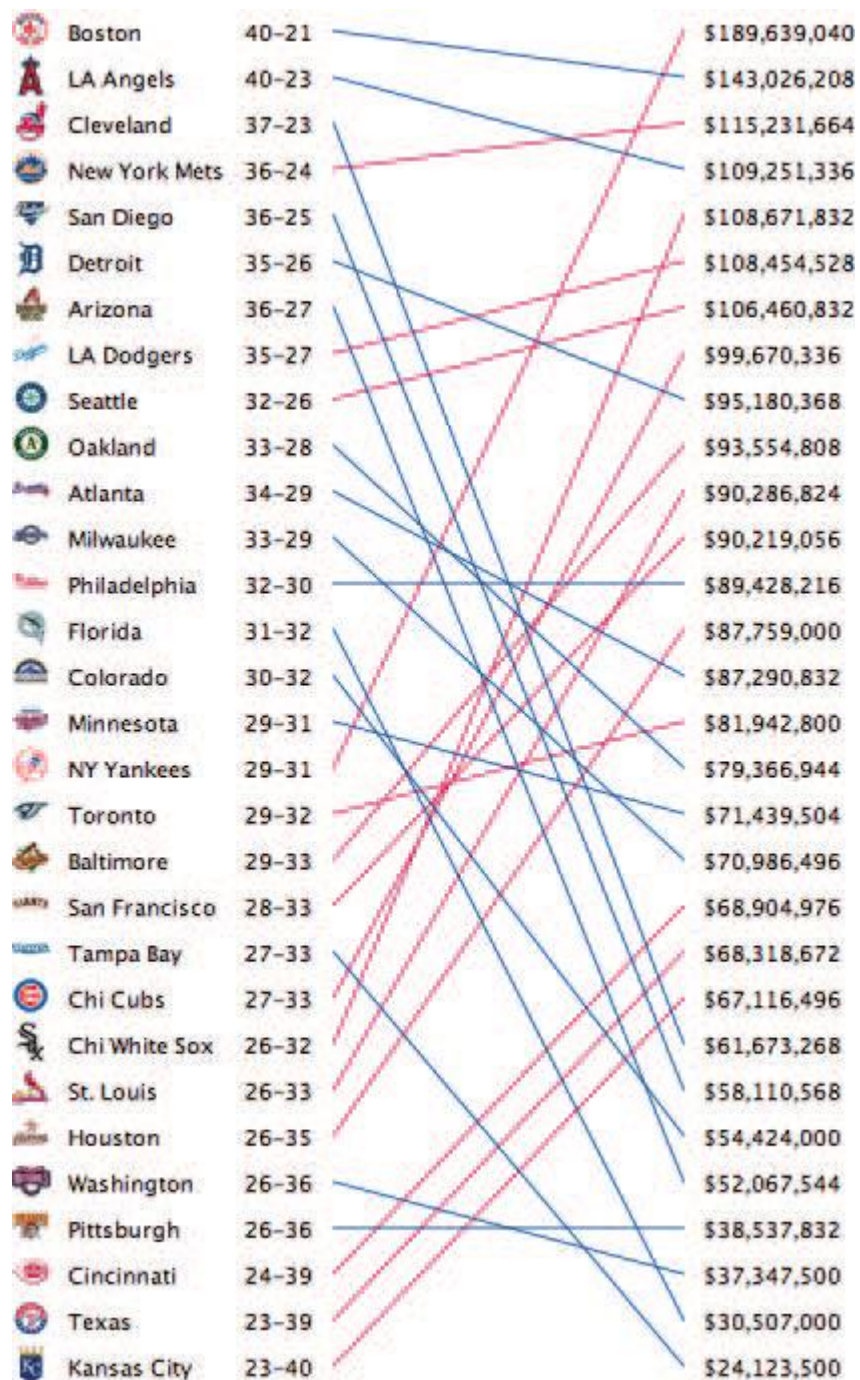
	Boston	40-21		\$189,639,040
	LA Angels	40-23		\$143,026,208
	Cleveland	37-23		\$115,231,664
	New York Mets	36-24		\$109,251,336
	San Diego	36-25		\$108,671,832
	Detroit	35-26		\$108,454,528
	Arizona	36-27		\$106,460,832
	LA Dodgers	35-27		\$99,670,336
	Seattle	32-26		\$95,180,368
	Oakland	33-28		\$93,554,808
	Atlanta	34-29		\$90,286,824
	Milwaukee	33-29		\$90,219,056
	Philadelphia	32-30		\$89,428,216
	Florida	31-32		\$87,759,000
	Colorado	30-32		\$87,290,832
	Minnesota	29-31		\$81,942,800
	NY Yankees	29-31		\$79,366,944
	Toronto	29-32		\$71,439,504
	Baltimore	29-33		\$70,986,496
	San Francisco	28-33		\$68,904,976
	Tampa Bay	27-33		\$68,318,672
	Chi Cubs	27-33		\$67,116,496
	Chi White Sox	26-32		\$61,673,268
	St. Louis	26-33		\$58,110,568
	Houston	26-35		\$54,424,000
	Washington	26-36		\$52,067,544
	Pittsburgh	26-36		\$38,537,832
	Cincinnati	24-39		\$37,347,500
	Texas	23-39		\$30,507,000
	Kansas City	23-40		\$24,123,500



## Highlighting the Lines

The first metric for the original question is whether teams are spending their money well. At its most basic, this is a yes or no question, so it will be important to highlight it as such with the representation. Teams spending their money well have a line that gets lower as it moves from left to right (connecting a high ranking in the standings to a low salary), whereas teams wasting money have lines that move upward from left to right. By using a color for each scenario, we can highlight the answer to the Boolean question of how well the team is performing. Color is a good choice in this case because we need only a pair of colors, and the detail being shown with the color is more important than any other feature in the diagram. To apply the colors, replace the `stroke(0)` line with the following:

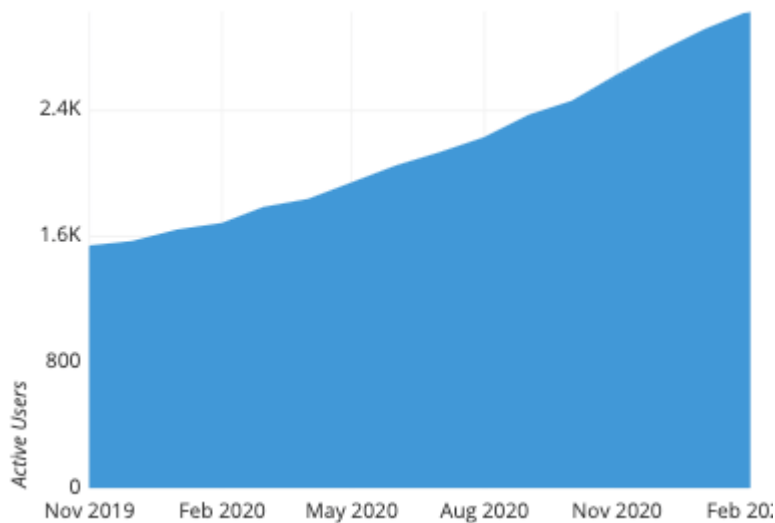
```
if (salaryY >= standingsY) {  
  stroke(33, 85, 156); // Blue for positive (or equal) difference.  
} else {  
  stroke(206, 0, 82); // Red for wasting money.  
}
```



# AREA CHART

## What is an area chart?

An area chart combines the [line chart](#) and [bar chart](#) to show how one or more groups' numeric values change over the progression of a second variable, typically that of time. An area chart is distinguished from a line chart by the addition of shading between lines and a baseline, like in a bar chart.



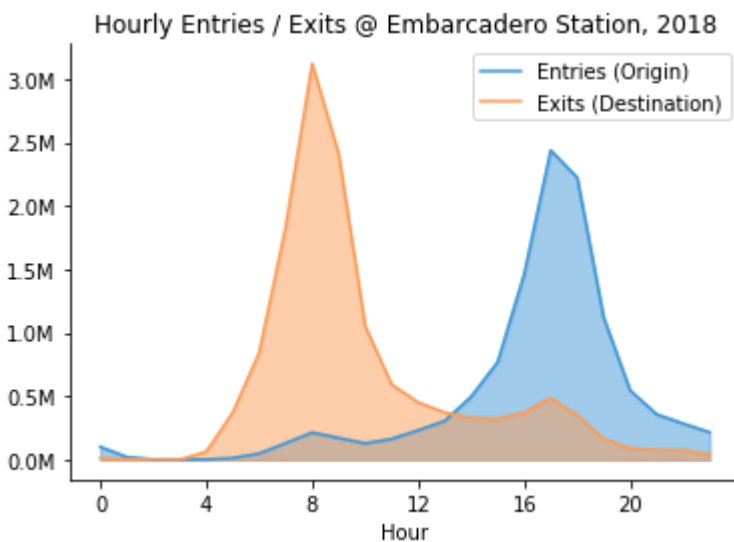
This area chart shows the number of active users for a fictional web-based company, computed by month. Values for each month can be measured not just from the vertical position of the top of the shape, but also the colored height between the baseline and top. In this chart, we can see that the number of active users has about doubled from November 2019 to February 2020, and that the rate of user gains has increased over time.

## When you should use an area chart

While the example above only plots a single line with shaded area, an area chart is typically used with multiple lines to make a comparison between groups (aka series) or to show how a whole is divided into component parts. This leads to two different types of area chart, one for each use case.

## **Overlapping area chart**

In the case that we want to compare the values between groups, we end up with an overlapping area chart. In an overlapping area chart, we start with a standard line chart. For each group, one point is plotted at each horizontal value with height indicating the group's value on the vertical axis variable; a line connects all of a group's points from left to right. The area chart adds shading between each line to a zero baseline. Since the shading for groups will usually overlap to some extent, some transparency is included in the shading so that all groups' lines can be easily seen. The shading helps to emphasize which group has the largest value based on which group's pure color is visible.

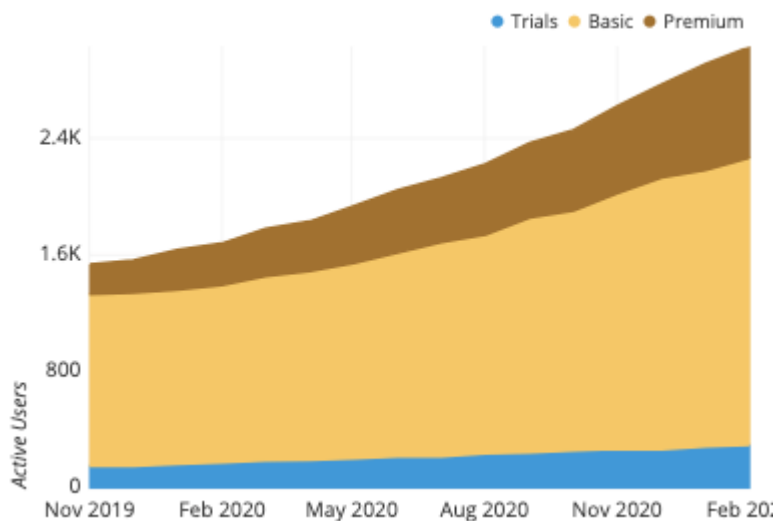


Be careful that one series is not always higher than the other, or the plot may become confused with the other type of area chart: the stacked area chart. In those cases, just keeping to the standard line chart will be a better choice.

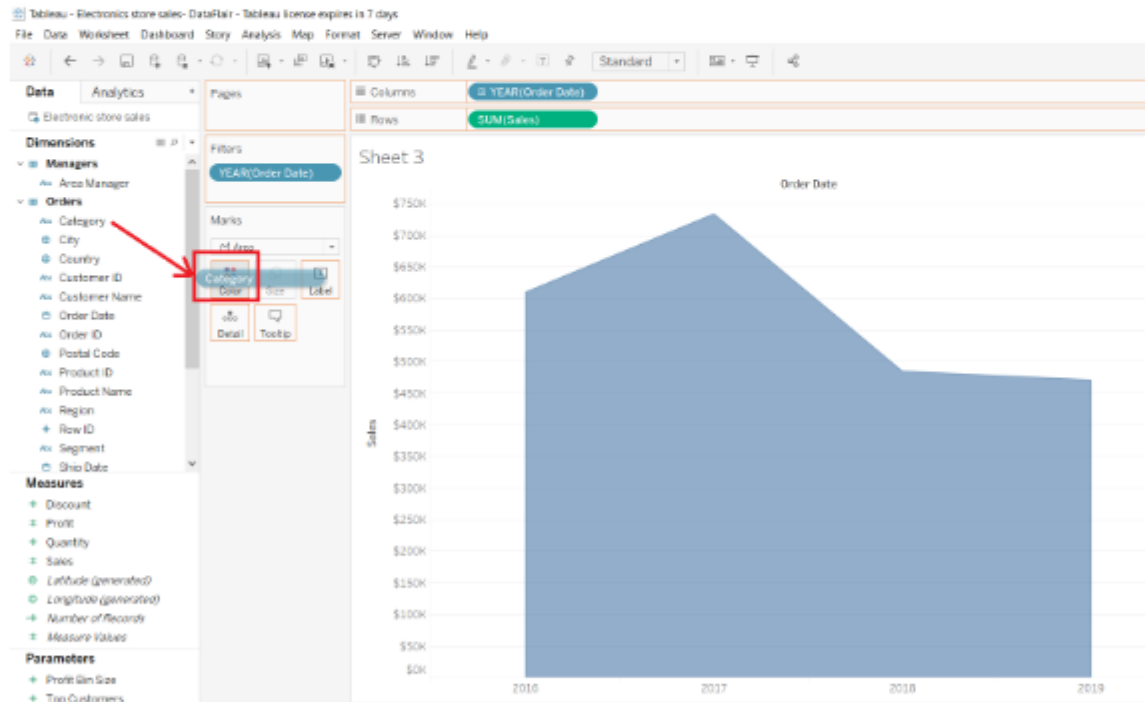
## **Stacked area chart**

Generally, when the term ‘area chart’ is used, what is actually implied is the stacked area chart. In the overlapping area chart, each line was shaded from its vertical value to a common baseline. In the stacked area chart, lines are plotted one at a time, with the height of the most recently-plotted group serving as a moving baseline. As such, the fully-stacked height of the topmost line will correspond to the total when summing across all groups.

You will use a stacked area chart when you want to track not only the total value, but also want to understand the breakdown of that total by groups. Comparing the heights of each segment of the curve allows us to get a general idea of how each subgroup compares to the other in their contributions to the total.



## Area chart in Tableau



## PIVOT TABLE

### What is a pivot table?

a pivot table provides an [interactive view of your data](#). With very little effort (and no formulas) you can look at the *same data from many different perspectives*. You can group data into categories, break down data into years and months, filter data to include or exclude categories, and even build charts.

*The beauty of pivot tables is they allow you to interactively explore your data in different ways.*

- Sample data
- Insert Pivot table
- Add fields
- Sort by value
- Refresh data
- Second value field
- Apply number formatting
- Group by date
- Add percent of total

## Sample data

The [sample data](#) contains 452 records with 5 fields of information: Date, Color, Units, Sales, and Region. This data is perfect for a pivot table.

B5    1/3/2016

	A	B	C	D	E	F	G	H	I	J
1										
2		<b>Sample sales data</b>								
3										
4		Date	Color	Region	Units	Sales				
5		3-Jan-16	Red	West	1	\$11.00				
6		13-Jan-16	Blue	South	8	\$96.00				
7		21-Jan-16	Green	West	2	\$26.00				
8		30-Jan-16	Blue	North	7	\$84.00				
9		7-Feb-16	Green	North	8	\$104.00				
10		13-Feb-16	Red	South	2	\$22.00				
11		21-Feb-16	Blue	East	5	\$60.00				
12		1-Mar-16	Green	West	2	\$26.00				
13		13-Mar-16	Blue	East	8	\$96.00				
14		23-Mar-16	Blue	North	7	\$84.00				
15		28-Mar-16	Green	West	2	\$26.00				
16		3-Apr-16	Blue	South	8	\$96.00				

Sample sales data  
in an Excel Table  
named "Table1"

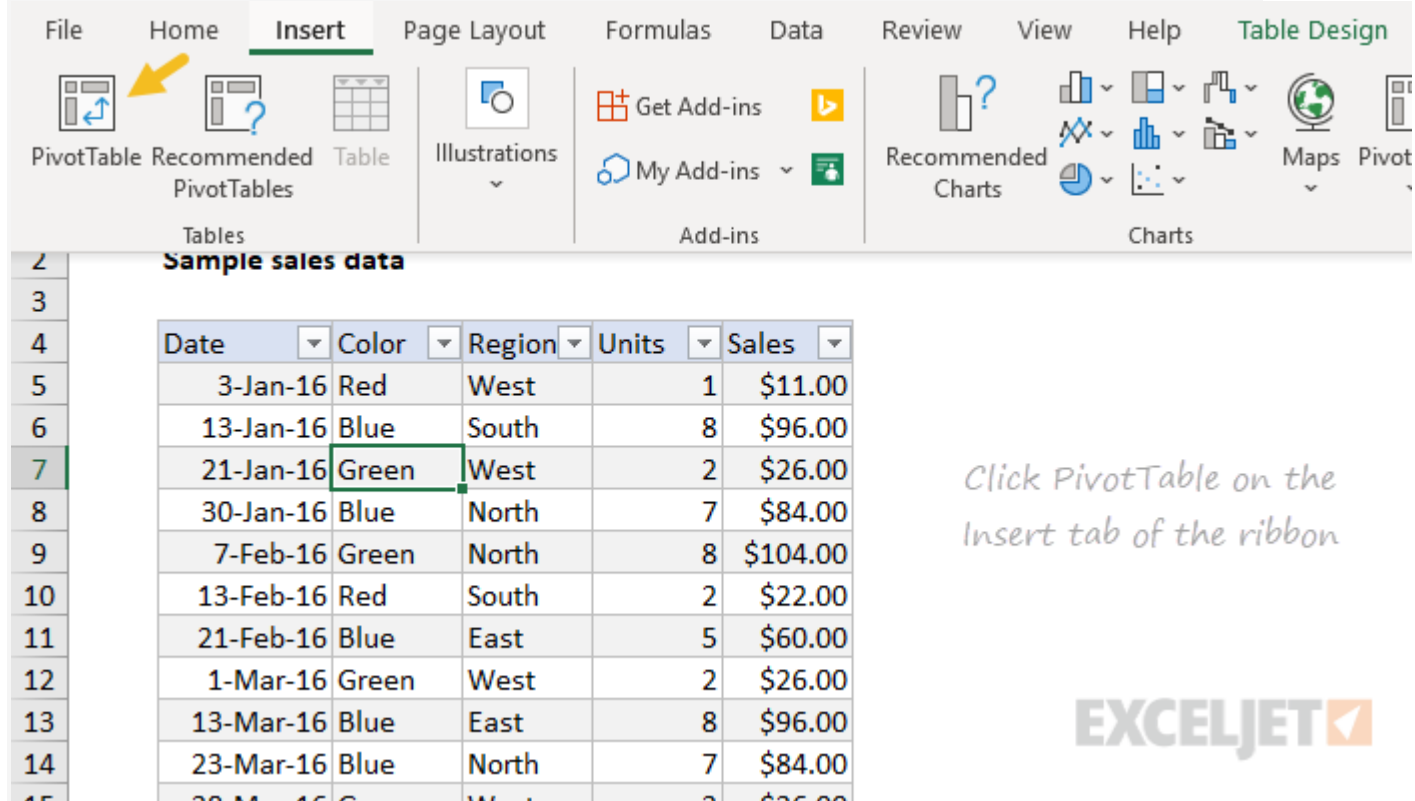
EXCELJET

Data in a proper [Excel Table](#) named "Table1". Excel Tables are a [great way to build pivot tables](#), because they automatically adjust as data is added or removed.

*Note: I know this data is very generic. But generic data is good for understanding pivot tables – you don't want to get tripped up on on a detail when learning the fun parts.*

## Insert Pivot Table

1. To start off, select *any cell in the data* and click Pivot Table on the Insert tab of the ribbon:



The screenshot shows the Excel ribbon with the 'Insert' tab selected. The 'PivotTable' icon is highlighted with a yellow arrow. Below the ribbon, a table of sample sales data is displayed. The cell containing 'Green' in the 'Color' column is selected.

Date	Color	Region	Units	Sales
3-Jan-16	Red	West	1	\$11.00
13-Jan-16	Blue	South	8	\$96.00
21-Jan-16	Green	West	2	\$26.00
30-Jan-16	Blue	North	7	\$84.00
7-Feb-16	Green	North	8	\$104.00
13-Feb-16	Red	South	2	\$22.00
21-Feb-16	Blue	East	5	\$60.00
1-Mar-16	Green	West	2	\$26.00
13-Mar-16	Blue	East	8	\$96.00
23-Mar-16	Blue	North	7	\$84.00

*Click PivotTable on the Insert tab of the ribbon*

EXCELJET

Excel will display the Create Pivot Table window. Notice the data range is already filled in. The default location for a new pivot table is New Worksheet.

2. Override the default location and enter H4 to place the pivot table on the current worksheet:



**Create PivotTable**

Choose the data that you want to analyze

☒ **Select a table or range**

Table/Range:

☐ **Use an external data source**

Connection name:

☐ **Use this workbook's Data Model**

Choose where you want the PivotTable report to be placed

☐ **New Worksheet**

☒ **Existing Worksheet**

Location:

Choose whether you want to analyze multiple tables

☐ **Add this data to the Data Model**

3. Click OK, and Excel builds an empty pivot table starting in cell H4.

H4

	A	B	C	D	E	F	G	H	I	J	K
1											
2											
3											
4											
5											
6											
7											
8											
9											
10											
11											
12											
13											
14											
15											
16											
17											
18											

**Sample sales data**

Date	Color	Region	Units	Sales
3-Jan-16	Red	West	1	\$11.00
13-Jan-16	Blue	South	8	\$96.00
21-Jan-16	Green	West	2	\$26.00
30-Jan-16	Blue	North	7	\$84.00
7-Feb-16	Green	North	8	\$104.00
13-Feb-16	Red	South	2	\$22.00
21-Feb-16	Blue	East	5	\$60.00
1-Mar-16	Green	West	2	\$26.00
13-Mar-16	Blue	East	8	\$96.00
23-Mar-16	Blue	North	7	\$84.00
28-Mar-16	Green	West	2	\$26.00
3-Apr-16	Blue	South	8	\$96.00
12-Apr-16	Green	South	1	\$13.00
16-Apr-16	Red	East	8	\$88.00

*New empty Pivot Table*

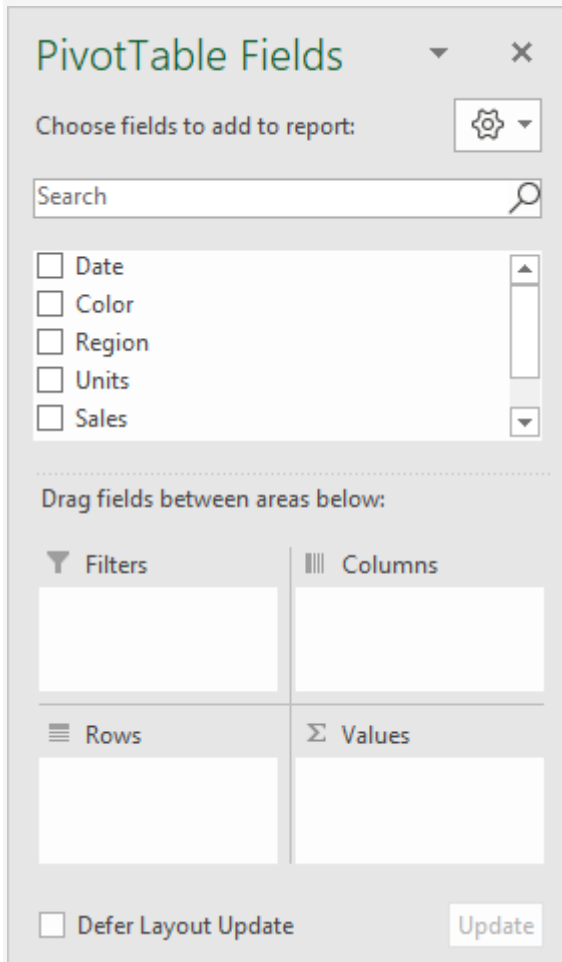
**PivotTable2**

To build a report, choose fields from the PivotTable Field List

EXCELJET

*Note: there are good reasons to place a pivot table on a different worksheet. However, when learning pivot tables, it's helpful to see both the source data and the pivot table at the same time.*

Excel also displays the PivotTable Fields pane, which is empty at this point. Note all five fields are listed, but unused:



To build a pivot table, drag fields into one the Columns, Rows, or Values area. The Filters area is used to apply global filters to a pivot table.

*Note: the pivot table fields pane shows how fields were used to create a pivot table. Learning to "read" the fields pane takes a bit of practice. See below and [also here](#) for more examples.*

## Add fields

1. Drag the Sales field to the Values area.

Excel calculates a grand total, 26356. This is the sum of all sales values in the entire data set:

H4																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																											
----	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

2. Drag the Color field to the Rows area.

Excel breaks out sales by Color. You can see Blue is the top seller, while Red comes in last:

H4										
	A	B	C	D	E	F	G	H	I	J
1										
2										
3										
4										
5										
6										
7										
8										
9										
10										
11										
12										
13										
14										
15										
16										

Sample sales data							
Date	Color	Region	Units	Sales		Color	Sum of Sales
3-Jan-16	Red	West	1	\$11.00		Blue	7464
13-Jan-16	Blue	South	8	\$96.00		Green	6414
21-Jan-16	Green	West	2	\$26.00		Red	5508
30-Jan-16	Blue	North	7	\$84.00		Silver	6970
7-Feb-16	Green	North	8	\$104.00		<b>Grand Total</b>	<b>26356</b>
13-Feb-16	Red	South	2	\$22.00			
21-Feb-16	Blue	East	5	\$60.00			
1-Mar-16	Green	West	2	\$26.00			
13-Mar-16	Blue	East	8	\$96.00			
23-Mar-16	Blue	North	7	\$84.00			
28-Mar-16	Green	West	2	\$26.00			
3-Apr-16	Blue	South	8	\$96.00			

Notice the Grand Total remains 26356. This makes sense, because we are still reporting on the full set of data.

Let's take a look at the fields pane at this point. You can see Color is a Row field, and Sales is a Value field:

EXCELJET

**PivotTable Fields** ▼ ✕

Choose fields to add to report: ⚙ ▼

Search 🔍

☐ Date  
☒ **Color**  
☐ Region  
☐ Units  
☒ **Sales**

Drag fields between areas below:

🔿 Filters	Columns

☰ Rows	Σ Values
Color ▼	Sum of Sales ▼

☐ Defer Layout Update      Update

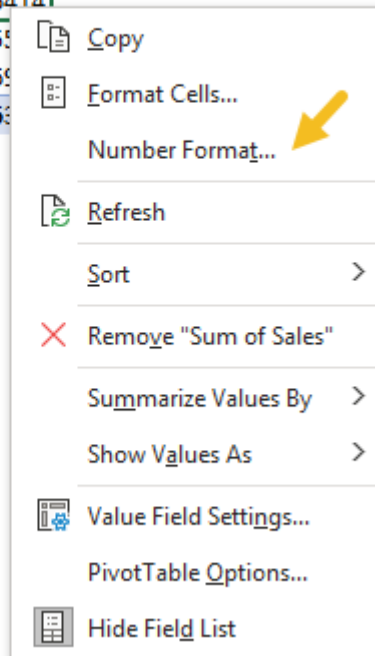
## Number formatting

Pivot Tables can apply and maintain number formatting automatically to numeric fields. This is a big time-saver when data changes frequently.

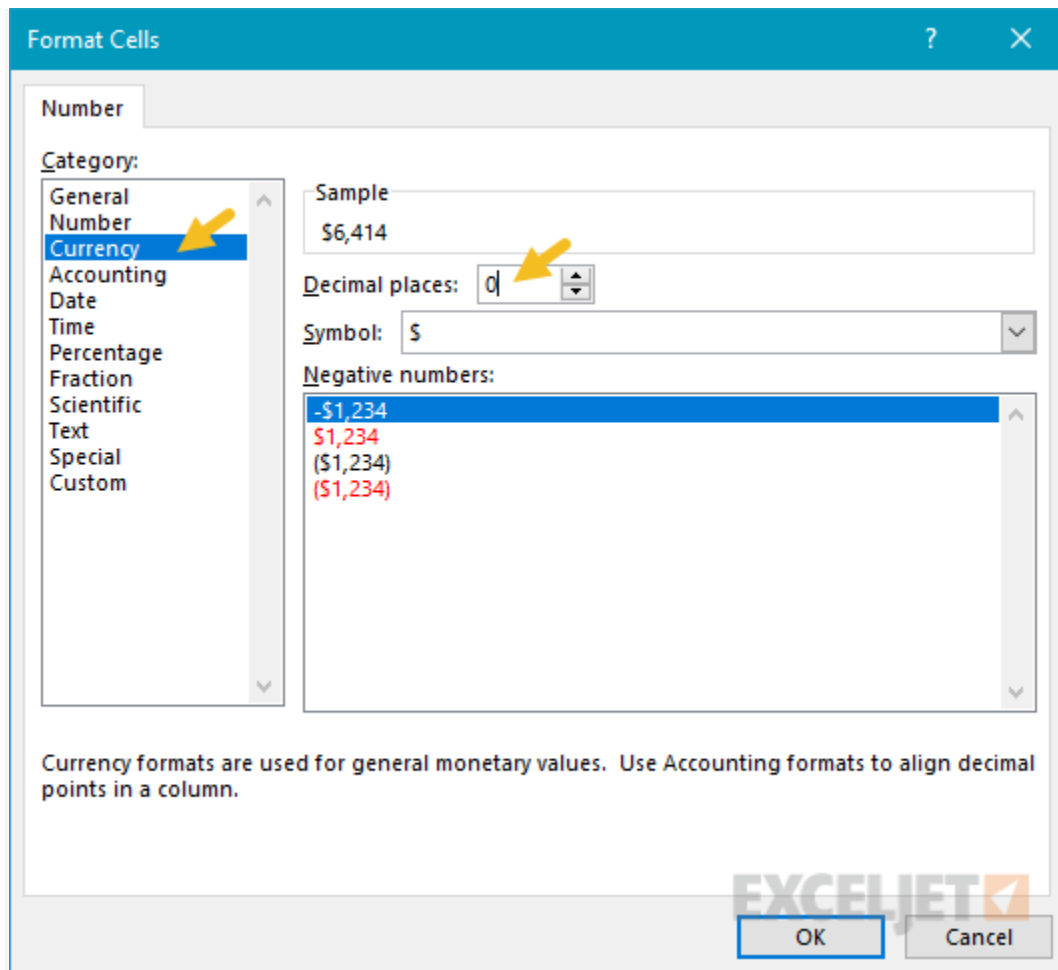
1. Right-click any Sales number and choose Number Format:

Color	Sum of Sales
Blue	7464
Green	6414
Red	59
Silver	69
<b>Grand Total</b>	<b>269</b>

Right-click,  
select Number  
Format



2. Apply Currency formatting with zero decimal places, then click OK:



In the resulting pivot table, all sales values have Currency format applied:

16										
	A	B	C	D	E	F	G	H	I	J
1										
2										
3										
4										
5										
6										
7										
8										
9										
10										
11										
12										
13										
14										
15										
16										

Sample sales data							
Date	Color	Region	Units	Sales		Color	Sum of Sales
3-Jan-16	Red	West	1	\$11.00		Blue	\$7,464
13-Jan-16	Blue	South	8	\$96.00		Green	\$6,414
21-Jan-16	Green	West	2	\$26.00		Red	\$5,508
30-Jan-16	Blue	North	7	\$84.00		Silver	\$6,970
7-Feb-16	Green	North	8	\$104.00		<b>Grand Total</b>	<b>\$26,356</b>
13-Feb-16	Red	South	2	\$22.00			
21-Feb-16	Blue	East	5	\$60.00			
1-Mar-16	Green	West	2	\$26.00			
13-Mar-16	Blue	East	8	\$96.00			
23-Mar-16	Blue	North	7	\$84.00			
28-Mar-16	Green	West	2	\$26.00			
3-Apr-16	Blue	South	8	\$96.00			

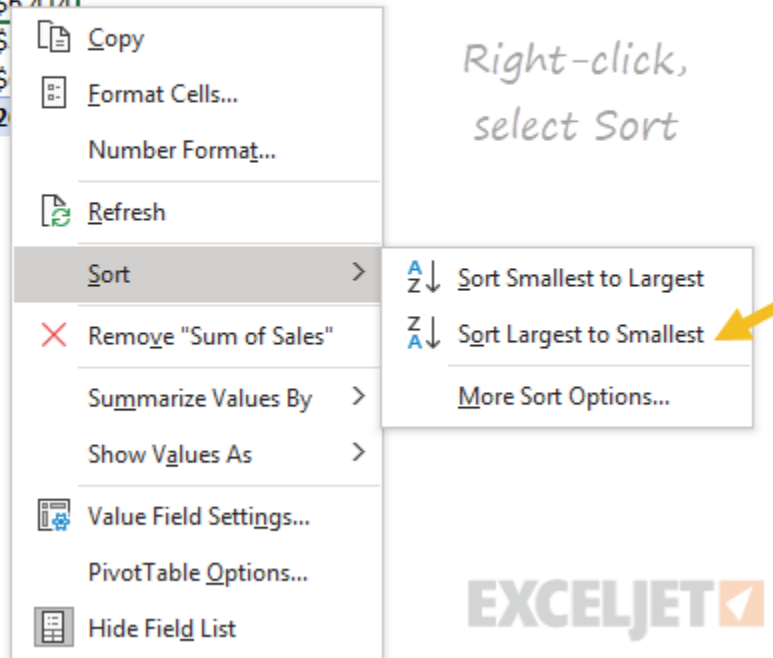
Currency format will continue to be applied to Sales values, even when the pivot table is reconfigured, or new data is added.

## Sorting by value

1. Right-click any Sales value and choose Sort > Largest to Smallest.



Color	Sum of Sales
Blue	\$7,464
Green	\$6,111
Red	\$
Silver	\$
Grand Total	\$2



Excel now lists top-selling colors first. This sort order will be maintained when data changes, or when the pivot table is reconfigured.

I5										
	A	B	C	D	E	F	G	H	I	J
1										
2										
3										
4										
5										
6										
7										
8										
9										
10										
11										
12										
13										
14										
15										
16										

Sample sales data

Sorted largest to smallest

Date	Color	Region	Units	Sales
3-Jan-16	Red	West	1	\$11.00
13-Jan-16	Blue	South	8	\$96.00
21-Jan-16	Green	West	2	\$26.00
30-Jan-16	Blue	North	7	\$84.00
7-Feb-16	Green	North	8	\$104.00
13-Feb-16	Red	South	2	\$22.00
21-Feb-16	Blue	East	5	\$60.00
1-Mar-16	Green	West	2	\$26.00
13-Mar-16	Blue	East	8	\$96.00
23-Mar-16	Blue	North	7	\$84.00
28-Mar-16	Green	West	2	\$26.00
3-Apr-16	Blue	South	8	\$96.00

Color	Sum of Sales
Blue	\$7,464
Silver	\$6,970
Green	\$6,414
Red	\$5,508
Grand Total	\$26,356



## Refresh data

Pivot table data needs to be "refreshed" in order to bring in updates. To reinforce how this works, we'll make a big change to the source data and watch it flow into the pivot table.

1. Select cell F5 and change \$11.00 to \$2000.
2. Right-click anywhere in the pivot table and select "Refresh".

	A	B	C	D	E	F	G	H	I	J
1										
2		Sample sales data						Right-click, select "Refresh"		
3										
4		Date	Color	Region	Units	Sales				
5		3-Jan-16	Red	West	1	\$2,000.00				
6		13-Jan-16	Blue	South	8	\$96.00				
7		21-Jan-16	Green	West	2	\$26.00				
8		30-Jan-16	Blue	North	7	\$84.00				
9		7-Feb-16	Green	North	8	\$104.00				
10		13-Feb-16	Red	South	2	\$22.00				
11		21-Feb-16	Blue	East	5	\$60.00				
12		1-Mar-16	Green	West	2	\$26.00				
13		13-Mar-16	Blue	East	8	\$96.00				
14		23-Mar-16	Blue	North	7	\$84.00				
15		28-Mar-16	Green	West	2	\$26.00				
16		3-Apr-16	Blue	South	8	\$96.00				
17		12-Apr-16	Green	South	1	\$13.00				
18		16-Apr-16	Red	East	8	\$88.00				
19		23-Apr-16	Red	West	6	\$66.00				
20		30-Apr-16	Green	South	5	\$65.00				

Color	Sum of Sales
Blue	\$7,464
Silver	\$6,970
Green	
Red	
Grand Total	

Copy
Format Cells...
Refresh
Sort
Filter
Subtotal "Color"
Expand/Collapse
Group...
Ungroup...

Notice "Red" is now the top selling color, and automatically moves to the top:

H5										
		Sample sales data								
		Date	Color	Region	Units	Sales				
		3-Jan-16	Red	West	1	\$2,000.00				
		13-Jan-16	Blue	South	8	\$96.00				
		21-Jan-16	Green	West	2	\$26.00				
		30-Jan-16	Blue	North	7	\$84.00				
		7-Feb-16	Green	North	8	\$104.00				
		13-Feb-16	Red	South	2	\$22.00				
		21-Feb-16	Blue	East	5	\$60.00				
		1-Mar-16	Green	West	2	\$26.00				
		13-Mar-16	Blue	East	8	\$96.00				
		23-Mar-16	Blue	North	7	\$84.00				
		28-Mar-16	Green	West	2	\$26.00				
		3-Apr-16	Blue	South	8	\$96.00				
		12-Apr-16	Green	South	1	\$13.00				
		16-Apr-16	Red	East	8	\$88.00				

Color	Sum of Sales
Red	\$7,497
Blue	\$7,464
Silver	\$6,970
Green	\$6,414
Grand Total	\$28,345

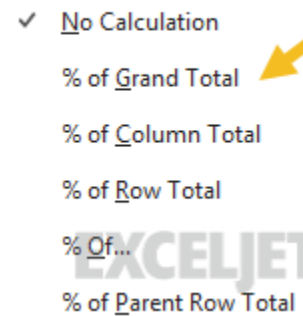
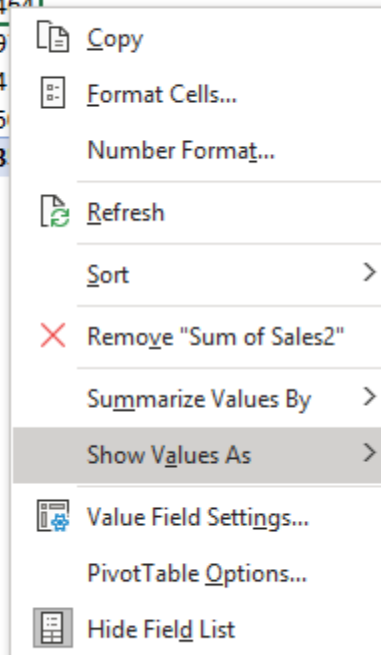
Red moves to the top as the best selling color



3. Right-click the second instance and choose "% of grand total":

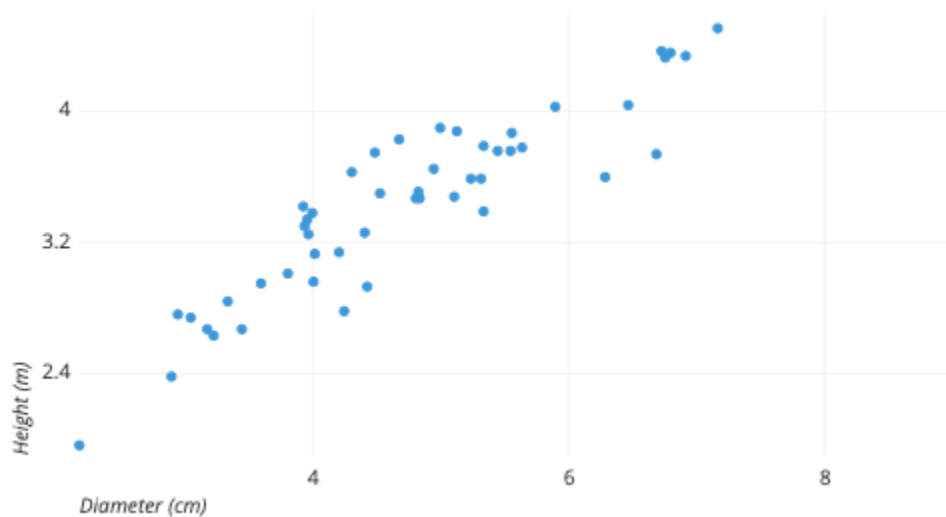
Color	Sum of Sales	Sum of Sales2
Blue	\$7,464	7464
Silver	\$6,970	6970
Green	\$6,414	6414
Red	\$5,508	5508
Grand Total	\$26,356	26356

*Changing  
calculation to show  
percent of total*



The result is a breakdown by color along with a percent of total:





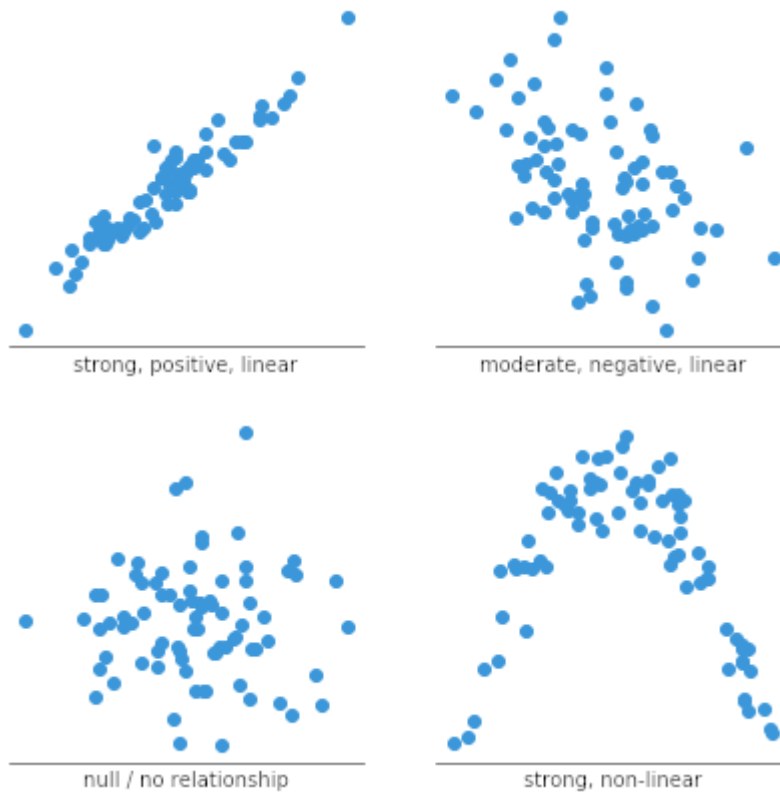
The example scatter plot above shows the diameters and heights for a sample of fictional trees. Each dot represents a single tree; each point's horizontal position indicates that tree's diameter (in centimeters) and the vertical position indicates that tree's height (in meters). From the plot, we can see a generally tight positive correlation between a tree's diameter and its height. We can also observe an outlier point, a tree that has a much larger diameter than the others. This tree appears fairly short for its girth, which might warrant further investigation.

## **When you should use a scatter plot**

Scatter plots' primary uses are to observe and show relationships between two numeric variables. The dots in a scatter plot not only report the values of individual data points, but also patterns when the data are taken as a whole.

Identification of correlational relationships are common with scatter plots. In these cases, we want to know, if we were given a particular horizontal value, what a good prediction would be for the vertical value. You will often see the variable on the horizontal axis denoted an independent variable, and the variable on the vertical axis the dependent variable. Relationships between

variables can be described in many ways: positive or negative, strong or weak, linear or nonlinear.



A scatter plot can also be useful for identifying other patterns in data. We can divide data points into groups based on how closely sets of points cluster together. Scatter plots can also show if there are any unexpected gaps in the data and if there are any outlier points. This can be useful if we want to segment the data into different parts, like in the development of user personas.





## Example of data structure

DIAMETER HEIGHT	
4.20	3.14
5.55	3.87
3.33	2.84
6.91	4.34
...	...

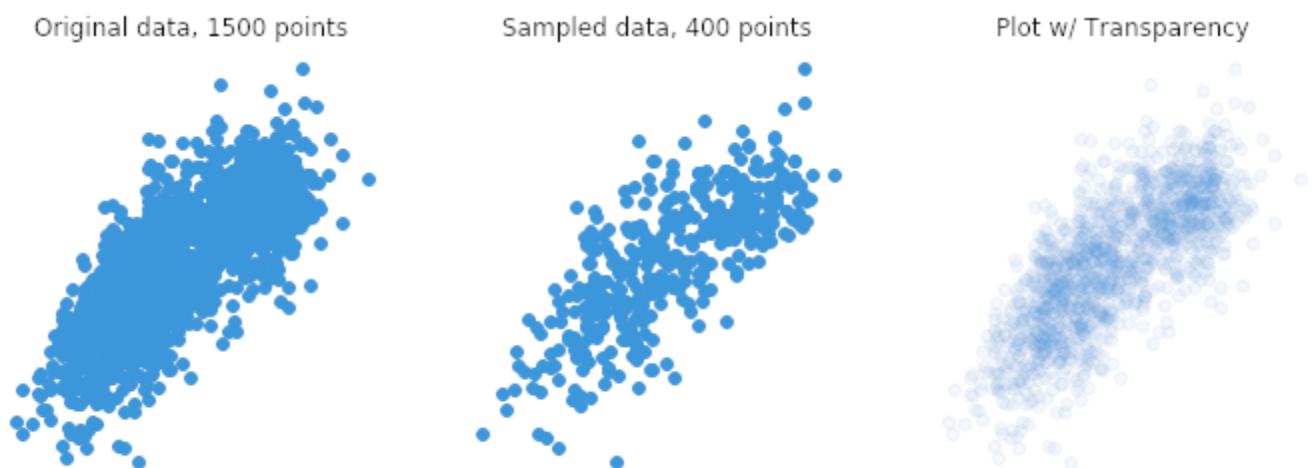
In order to create a scatter plot, we need to select two columns from a data table, one for each dimension of the plot. Each row of the table will become a single dot in the plot with position according to the column values.

## Common issues when using scatter plots

## **Overplotting**

When we have lots of data points to plot, this can run into the issue of overplotting. Overplotting is the case where data points overlap to a degree where we have difficulty seeing relationships between points and variables. It can be difficult to tell how densely-packed data points are when many of them are in a small area.

There are a few common ways to alleviate this issue. One alternative is to sample only a subset of data points: a random selection of points should still give the general idea of the patterns in the full data. We can also change the form of the dots, adding transparency to allow for overlaps to be visible, or reducing point size so that fewer overlaps occur. As a third option, we might even choose a different chart type like the [heatmap](#), where color indicates the number of points in each bin. Heatmaps in this use case are also known as 2-d histograms.



## **Interpreting correlation as causation**

This is not so much an issue with creating a scatter plot as it is an issue with its interpretation. Simply because we observe a relationship between two

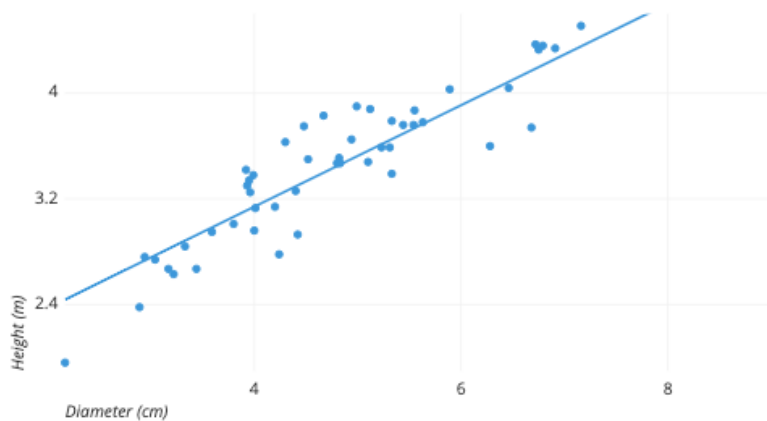
variables in a scatter plot, it does not mean that changes in one variable are responsible for changes in the other. This gives rise to the common phrase in statistics that [correlation does not imply causation](#). It is possible that the observed relationship is driven by some third variable that affects both of the plotted variables, that the causal link is reversed, or that the pattern is simply coincidental.

For example, it would be wrong to look at city statistics for the amount of green space they have and the number of crimes committed and conclude that one causes the other, this can ignore the fact that larger cities with more people will tend to have more of both, and that they are simply correlated through that and other factors. If a causal link needs to be established, then further analysis to control or account for other potential variables effects needs to be performed, in order to rule out other possible explanations.

## **Common scatter plot options**

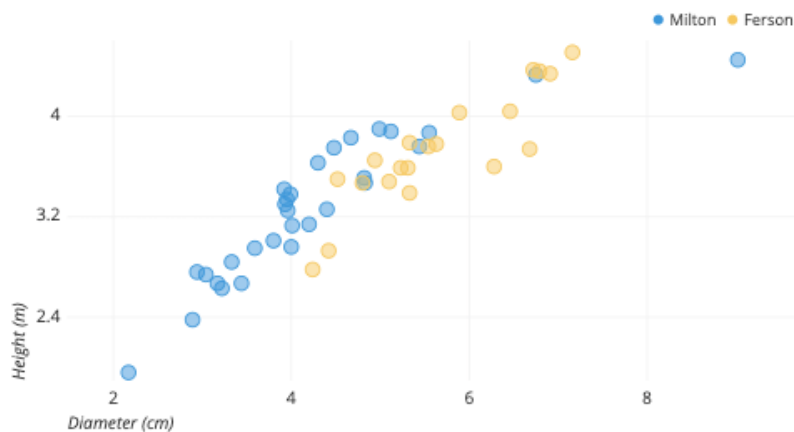
### **Add a trend line**

When a scatter plot is used to look at a predictive or correlational relationship between variables, it is common to add a trend line to the plot showing the mathematically best fit to the data. This can provide an additional signal as to how strong the relationship between the two variables is, and if there are any unusual points that are affecting the computation of the trend line.



## **Categorical third variable**

A common modification of the basic scatter plot is the addition of a third variable. Values of the third variable can be encoded by modifying how the points are plotted. For a third variable that indicates categorical values (like geographical region or gender), the most common encoding is through point color. Giving each point a distinct hue makes it easy to show membership of each point to a respective group.



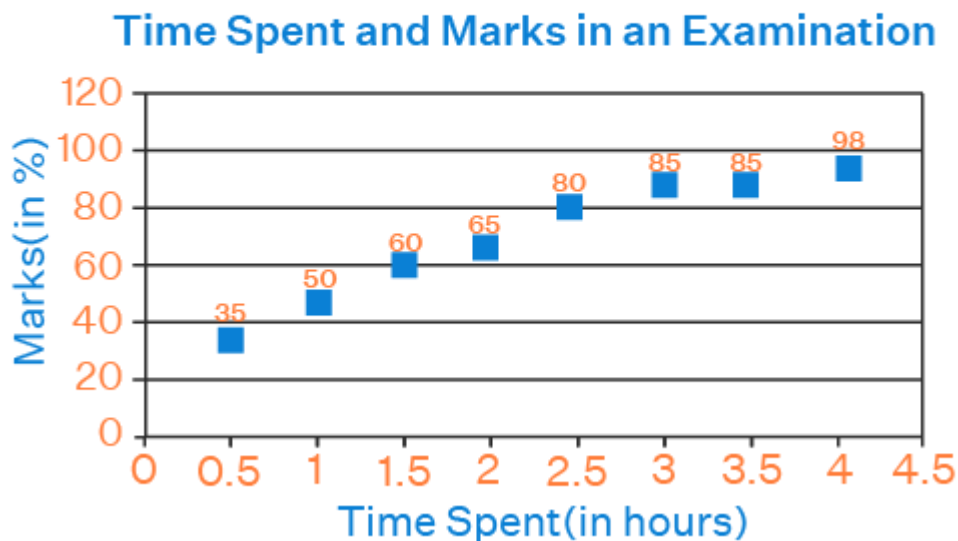
## Types of Scatter Plot

A scatter plot helps find the relationship between two variables. This relationship is referred to as a correlation. Based on the correlation, scatter plots can be classified as follows.

- Scatter Plot for Positive Correlation
- Scatter Plot for Negative Correlation
- Scatter Plot for Null Correlation

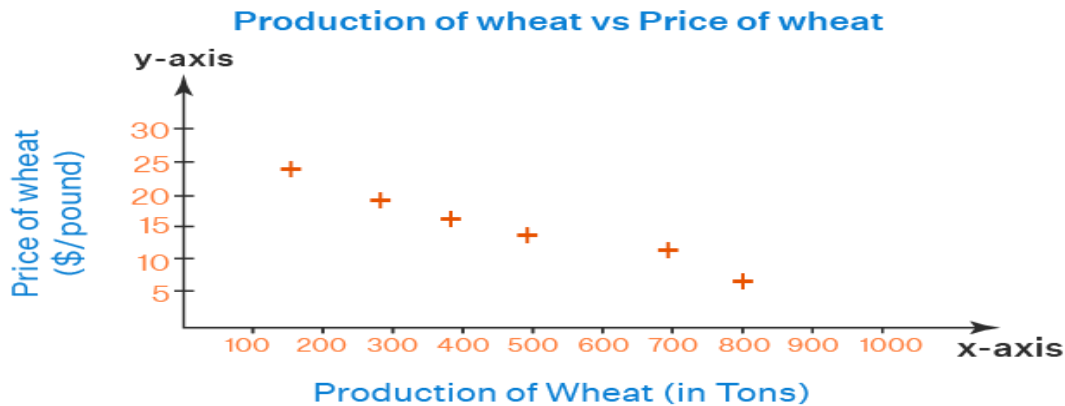
### Scatter Plot for Positive Correlation

A scatter plot with increasing values of both variables can be said to have a positive correlation. The scatter plot for the relationship between the time spent studying for an examination and the marks scored can be referred to as having a positive correlation.



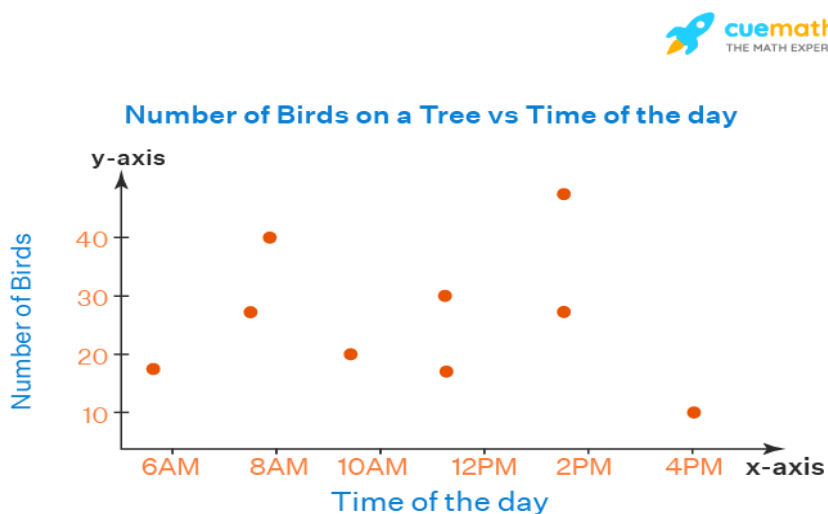
### Scatter Plot for Negative Correlation

A scatter plot with an increasing value of one variable and a decreasing value for another variable can be said to have a negative correlation. Observe the below image of negative scatter plot depicting the amount of production of wheat against the respective price of wheat.



## Scatter Plot for Null Correlation

A scatter plot with no clear increasing or decreasing trend in the values of the variables is said to have no correlation. Here the points are distributed randomly across the graph. For example, the data for the number of birds on a tree at different times of the day does not show any correlation. Observe the below scatter plot showing the number of birds on a tree versus time of the day.



# TREE MAP

## Description

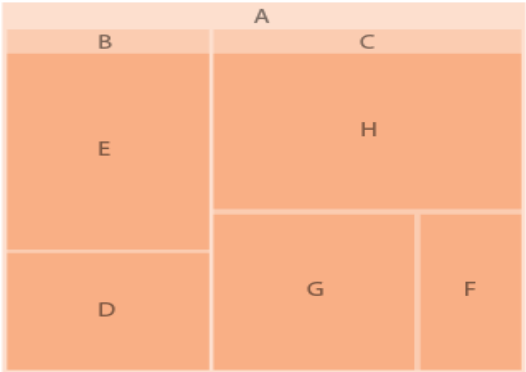
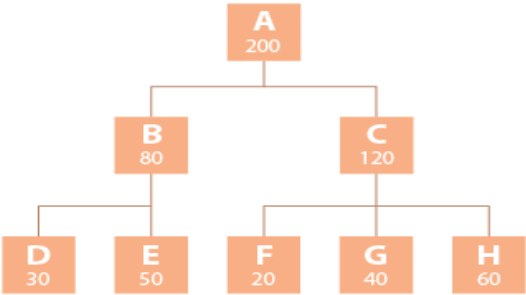
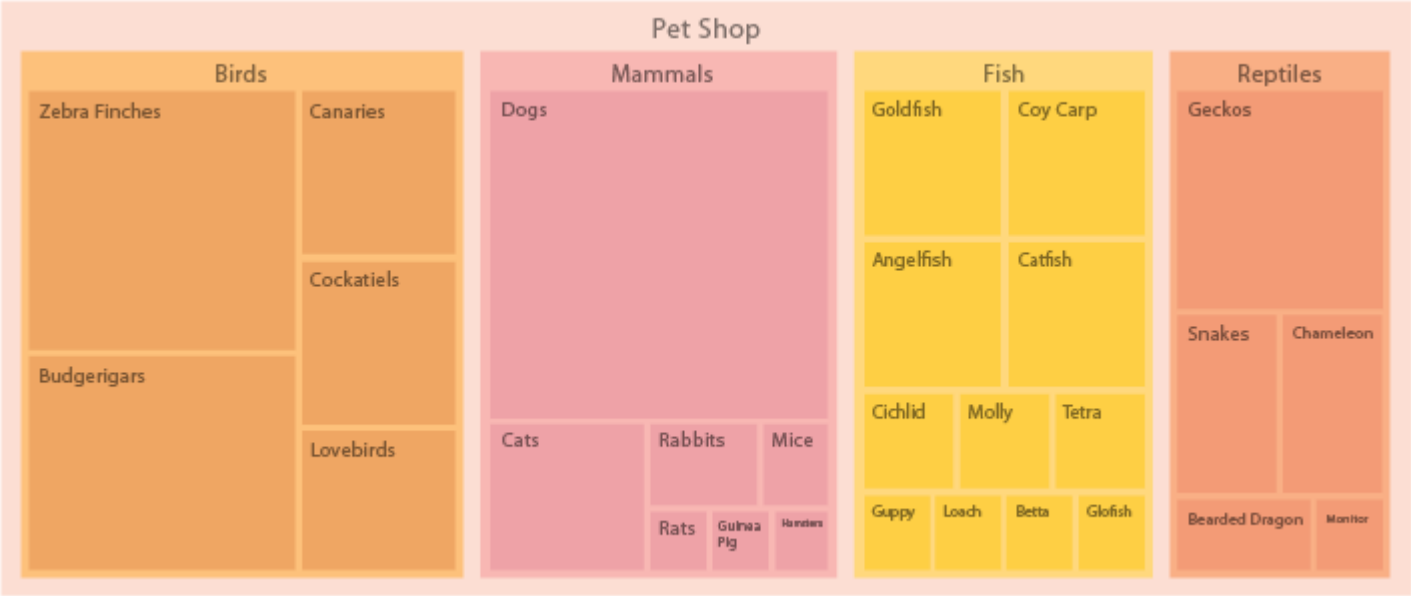
Treemaps are an alternative way of visualising the hierarchical structure of a Tree Diagram while also displaying quantities for each category via area size. Each category is assigned a rectangle area with their subcategory rectangles nested inside of it.

When a quantity is assigned to a category, its area size is displayed in proportion to that quantity and to the other quantities within the same parent category in a part-to-whole relationship. Also, the area size of the parent category is the total of its subcategories. If no quantity is assigned to a subcategory, then it's area is divided equally amongst the other subcategories within its parent category.

The way rectangles are divided and ordered into sub-rectangles is dependent on the tiling algorithm used. Many tiling algorithms have been developed, but the "squarified algorithm" which keeps each rectangle as square as possible is the one commonly used.

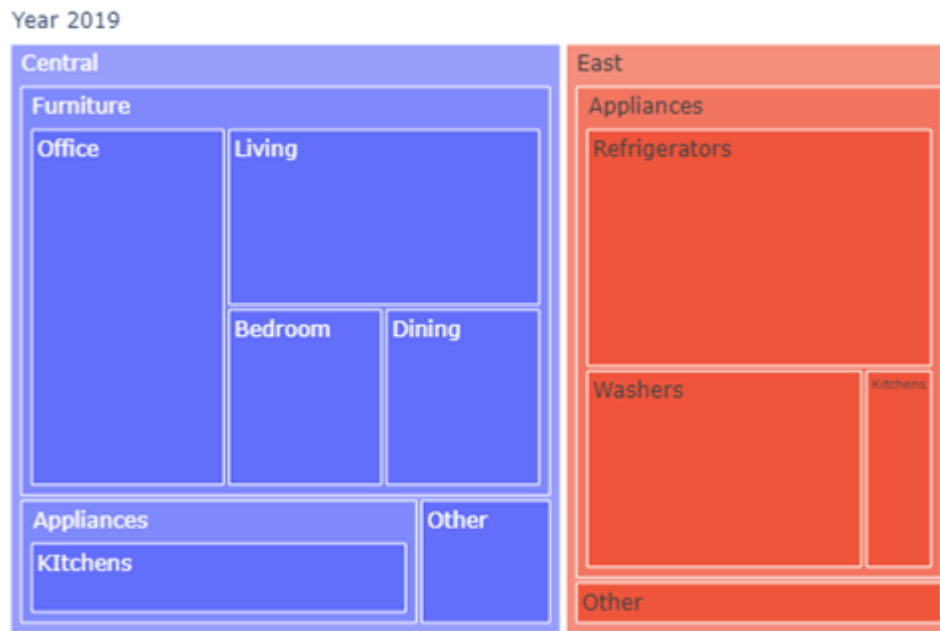
Ben Shneiderman originally developed Treemaps as a way of visualising a vast file directory on a computer, without taking up too much space on the screen. This makes Treemaps a more compact and space-efficient option for displaying hierarchies, that gives a quick overview of the structure. Treemaps are also great at comparing the proportions between categories via their area size.

The downside to a Treemap is that it doesn't show the hierarchal levels as clearly as other charts that visualise hierarchal data (such as a Tree Diagram or Sunburst Diagram).





Treemap is a **rectangle-based visualization** that allows you to represent a hierarchically-ordered (tree-structured) set of data. The conceptual idea is to compare quantities and show patterns of some hierarchical structure in a physically restricted space. For that purpose, rectangles of different sizes and colors are used to display the dataset from different perspectives. The goal is not to indicate the exact numerical values, but to “break” the dataset into its constituent parts and quickly identify its larger and smaller components.



## Tree map using Tableau

To create a treemap that shows aggregated sales totals across a range of product categories, follow the steps below.

1. Connect to the **Sample - Superstore** data source.

2. Drag the **Sub-Category** dimension to **Columns**.

A horizontal axis appears, which shows product categories.

3. Drag the **Sales** measure to **Rows**.

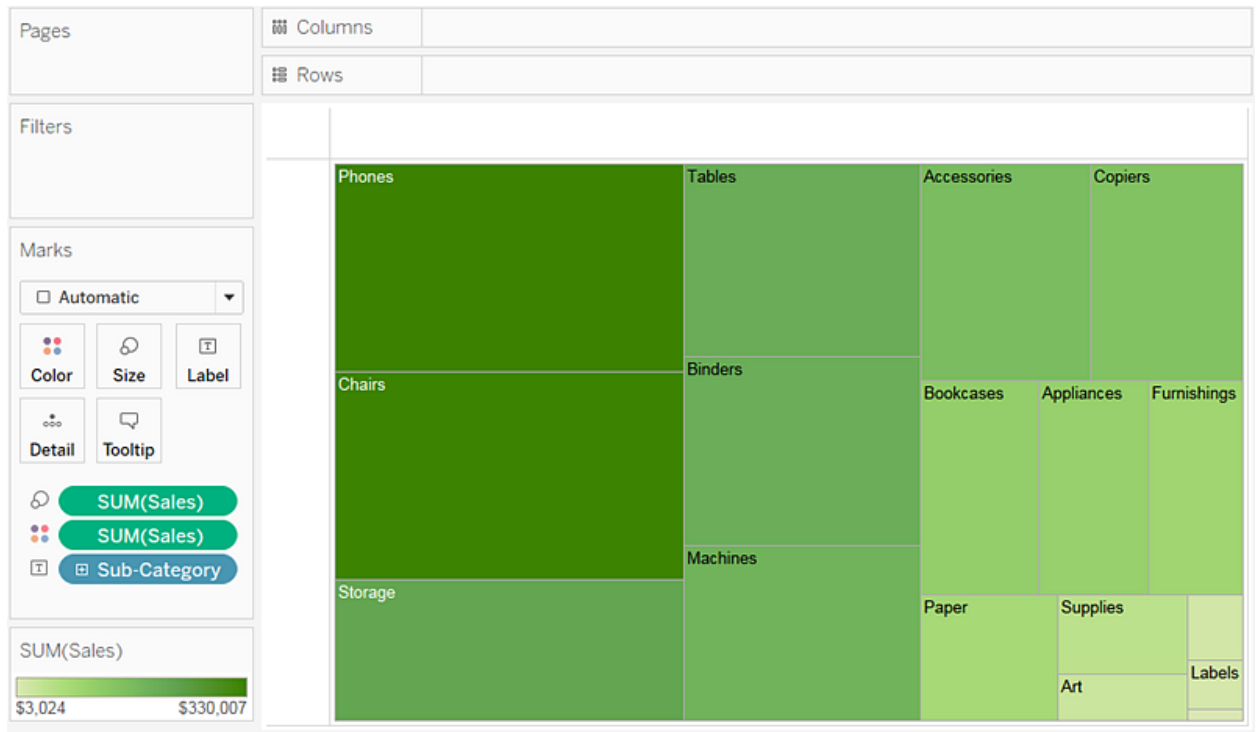
Tableau aggregates the measure as a sum and creates a vertical axis.

Tableau displays a bar chart—the default chart type when there is a dimension on the **Columns** shelf and a measure on the **Rows** shelf.

4. Click **Show Me** on the toolbar, then select the treemap chart type.

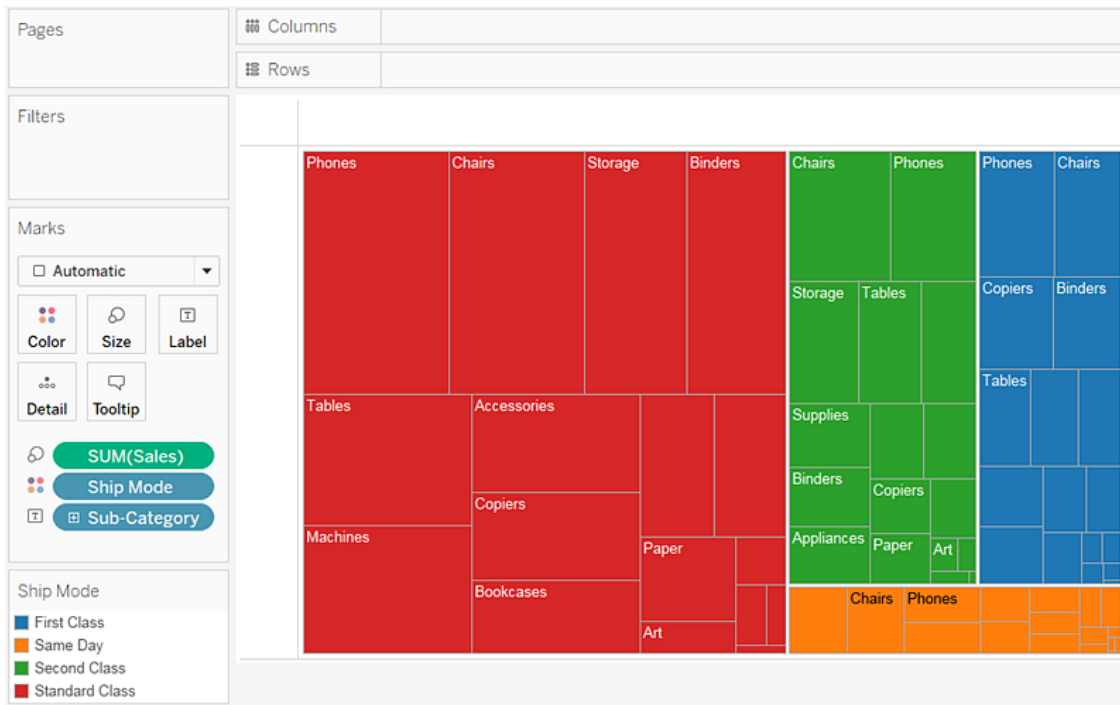


Tableau displays the following treemap:



In this treemap, both the size of the rectangles and their color are determined by the value of **Sales**—the greater the sum of sales for each category, the darker and larger its box.

5. Drag the **Ship Mode** dimension to **Color** on the **Marks** card. In the resulting view, **Ship Mode** determines the color of the rectangles—and sorts them into four separate areas accordingly. **Sales** determines the size of the rectangles:



## Ideal use cases for a treemap chart

Like every chart type and data visualization technique, the treemap charts works well only if it is used in situations that justify its use. Let's take a look at what are the ideal use cases that warrant the use of a treemap chart:

**Your data needs to be studied w.r.t two quantitative values.** Each rectangle (node) in the treemap chart showcases the values for two quantitative values. Like said above, the dimensions of the rectangles in the sample treemap chart (above) represent the units sold for a model in the current year and the color represents the growth in sales w.r.t the previous year sales.

**You have a very large amount of hierarchical data and a space constraint.** Treemap charts are equipped to be able to plot more than tens of thousands of data points. There are other charts that can be used for plotting hierarchical data; some of the charts that may quickly come to mind are the multi-level pie chart and the drag-node chart. However, these charts present a space constraint as the number of data points increases beyond a certain limit. Additionally, the multi-level

pie chart is circular while the treemap is linear; a linear chart is easier to read and understand than a circular one.

**You want a quick, high level summary of the similarities and anomalies within one category as well as between multiple categories.** The dimensions and colors of the rectangles (nodes) in a treemap chart are configured based on the numerical values assigned to each node. This makes it easy to identify the trends and patterns between the nodes of all the categories plotted on the chart as well between the nodes of a single category. For example, Honda-manufactured bikes have done better in the Street category than in any other category.

**Your data can be organized at several levels.** Treemap charts support the drill down feature; chart users can easily drill down to do a detailed study of data at several granular levels.

## CHAPTER 8

# Visualization Techniques for Trees, Graphs, and Networks

While most of the visualization techniques discussed thus far focus on the display of data values and their attributes, another important application of visualization is the conveying of relational information, e.g., how data items or records are related to each other. These interrelationships can take many forms:

- part/subpart, parent/child, or other hierarchical relation;
- connectedness, such as cities connected by roads or computers connected by networks;
- derived from, as in a sequence of steps or stages;
- shared classification;
- similarities in values;
- similarities in attributes (e.g., spatial, temporal).

Relationships can be simple or complex: unidirectional or bi-directional, nonweighted or weighted, certain or uncertain. Indeed, the relationships may provide more and richer information than that contained in the data records. Applications for visualizing relational information are equally diverse, from categorizing biological species, to exploring document archives, to studying a terrorist network.

In this chapter we will examine a number of techniques that have been developed for visualizing relational information. This presentation, however, will just be the tip of the iceberg, as tree and graph visualization is a well-established field, with its own books, journals, conferences, software packages, and algorithms.

## 8.1 Displaying Hierarchical Structures

Trees or hierarchies (we'll use the terms interchangeably) are one of the most common structures to hold relational information. For this reason, many visualization techniques have been developed for display of such information. We can divide these techniques into two classes of algorithms: space-filling and non-space-filling. The rest of this section will provide details on implementing algorithms for visualizing this type of data.

### 8.1.1 Space-Filling Methods

As the name implies, space-filling techniques make maximal use of the display space. This is accomplished by using juxtapositioning to imply relations, as opposed to, for example, conveying relations with edges joining data objects. The two most common approaches to generating space-filling hierarchies are rectangular and radial layouts.

Treemaps [176] and their many variants are the most popular form of rectangular space-filling layout. In the basic treemap, a rectangle is recursively divided into slices, alternating horizontal and vertical slicing, based on the populations of the subtrees at a given level. Pseudocode for this process is given in Figure 8.1, and an example is shown in Figure 8.2.

As mentioned, many variants on treemaps have been proposed and developed since they were introduced, including *squarified treemaps* [40] (to reduce the occurrence of long, thin rectangles) and nested *treemaps* [176] (to emphasize the hierarchical structure).

The methods described above are structured using horizontal and vertical divisions to convey the hierarchy. A number of other approaches are possible, however, such as those that divide space radially. Radial space-filling *hierarchy visualizations*, sometimes referred to as sunburst displays [336], have the root of the hierarchy in the center of the display and use nested rings to convey the layers of the hierarchy. Each ring is divided based on the number of nodes at that level. These techniques follow a similar strategy to treemaps, in that the number of terminal nodes in a subtree determines

```

Start: Main Program
  Width = width of rectangle
  Height = height of rectangle
  Node = root node of the tree
  Origin = position of rectangle, e.g., [0,0]
  Orientation = direction of cuts, alternating between horizontal and vertical
  Treemap(Node, Orientation, Origin, Width, Height)
End: Main Program

Treemap(node n, orientation o, position orig, hsize w, vsize h)
  if n is a terminal node (i.e., it has no children)
    draw-rectangle(orig, w, h)
    return
  for each child of n (child_i), get number of terminal nodes in subtree
  sum up number of terminal nodes
  compute percentage of terminal nodes in n from each subtree (percent-i)
  if orientation is horizontal
    for each subtree
      compute offset of origin based on origin and width (offset-i)
      treemap(child_i, vertical, orig + offset-i, w * percent-i, h)
  else
    for each subtree
      compute offset of origin based on origin and height (offset-i)
      treemap(child_i, horizontal, orig + offset-i, w, h * percent-i)
End: Treemap

```

Figure 8.1. Pseudocode for drawing a hierarchy using a treemap.

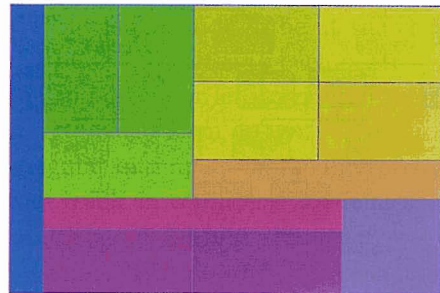
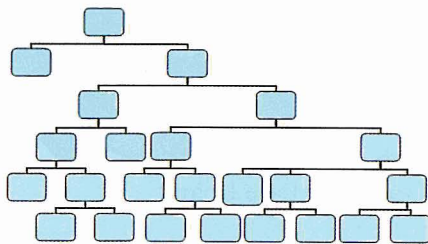


Figure 8.2. A sample hierarchy and the corresponding treemap display.



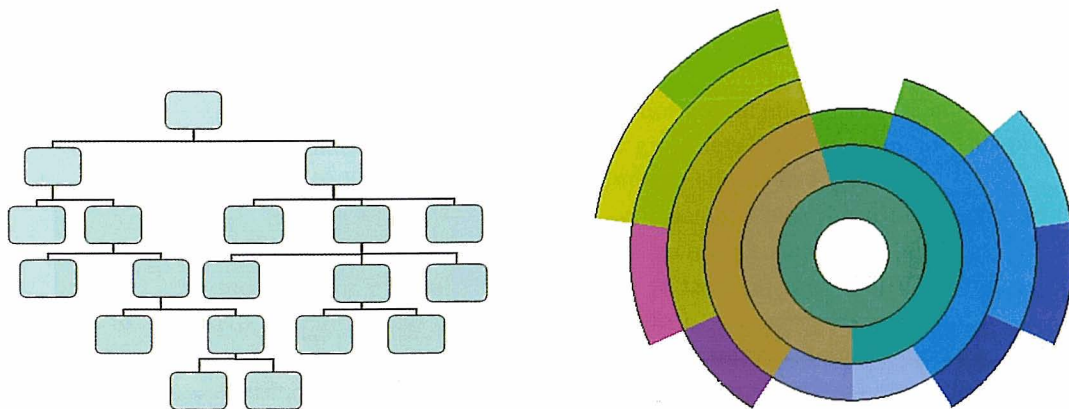
```

Start: Main Program
  Start = start angle for a node (initially 0)
  End = end angle for a node (initially 360)
  Origin = position of center of sunburst, e.g., [0,0]
  Level = current level of hierarchy (initially 0)
  Width = thickness of each radial band - based on max depth and display size
  Sunburst(Node, Start, End, Level)
End: Main Program

Sunburst(node n, angle st, angle en, level l)
  if n is a terminal node (i.e., it has no children)
    draw_radial_section(Origin, st, en, l * Width, (l+1) * Width)
    return
  for each child of n (child-i), get number of terminal nodes in subtree
  sum up number of terminal nodes
  compute percentage of terminal nodes in n from each subtree (percent_i)
  for each subtree
    compute start/end angle based on size of subtrees, order, and angle range
    Sunburst(child-i, st_i, en_i, l+1)
End: Sunburst

```

**Figure 8.3.** Pseudocode for drawing a hierarchy using a sunburst display.



**Figure 8.4.** A sample hierarchy and the corresponding sunburst display.

the amount of screen space that will be allocated for it. However, unlike treemaps, which assign most screen space to conveying the terminal nodes, radial techniques also show the intermediate nodes. The process is described in pseudocode in Figure 8.3, and an example is shown in Figure 8.4.

For these and other space-filling techniques, color can be used to convey many attributes, such as a value associated with the node (e.g., classification) or it may reinforce the hierarchical relationships, e.g., siblings and parents may have similarities in color, as seen in Figure 8.4. Symbols and other markings may also be embedded in the rectangular or circular segments to communicate other data features.

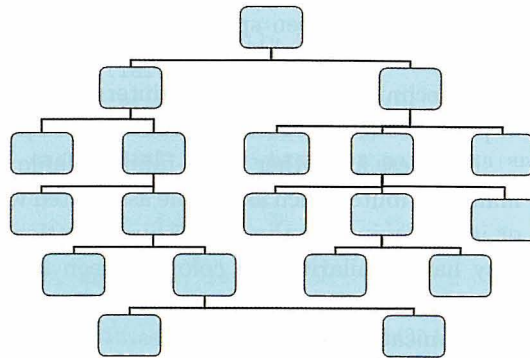
### 8.1.2 Non-Space-Filling Methods

The most common representation used to visualize tree or hierarchical relationships is a *node-link diagram*. Organizational charts, family trees, and tournament pairings are just some of the common applications for such diagrams. The drawing of such trees is influenced the most by two factors: the fan-out degree (e.g., the number of siblings a parent node can have) and the depth (e.g., the furthest node from the root). Trees that are significantly constrained in one or both of these aspects, such as a binary tree or a tree with only three or four levels, tend to be much easier to draw than those with fewer constraints.

When designing an algorithm for drawing any node-link diagram (not just trees), one must consider three categories of often-contradictory guidelines: drawing conventions, constraints, and aesthetics. Conventions may include restricting edges to be either a single straight line, a series of rectilinear lines, polygonal lines, or curves. Other conventions might be to place nodes on a fixed grid, or to have all sibling nodes share the same vertical position. Constraints may include requiring a particular node to be at the center of the display, or that a group of nodes be located close to each other, or that certain links must either go from top to bottom or left to right. Each of the above guidelines can be used to drive the algorithm design.

Aesthetics, however, often have significant impact on the interpretability of a tree or graph drawing, yet often result in conflicting guidelines. Some typical aesthetic rules include:

- minimize line crossings
- maintain a pleasing aspect ratio
- minimize the total area of the drawing



**Figure 8.5.** An example of visualizing hierarchies with a simple node-link diagram, using equal spacing per level.

- minimize the total length of the edges
- minimize the number of bends in the edges
- minimize the number of distinct angles or curvatures used
- strive for a symmetric structure

For trees, especially balanced ones, it is relatively easy to design algorithms that adhere to many, if not most, of these guidelines. For example, a simple tree drawing procedure is given below (sample output is shown in Figure 8.5):

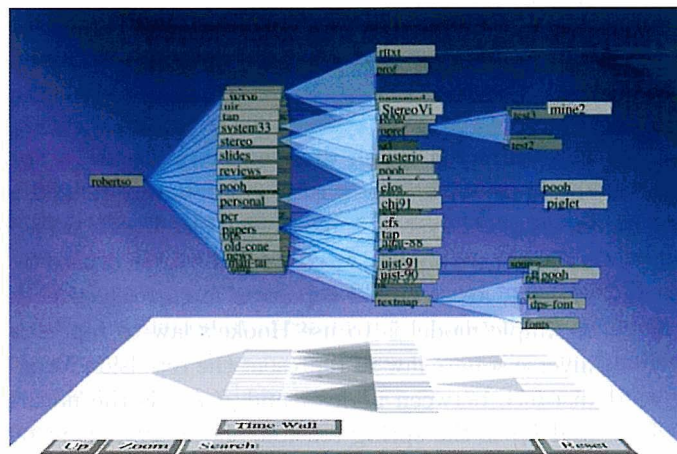
1. Slice the drawing area into equal-height slabs, based on the depth of the tree.
2. For each level of the tree, determine how many nodes need to be drawn.
3. Divide each slice into equal-sized rectangles based on the number of nodes at that level.
4. Draw each node in the center of its corresponding rectangle.
5. Draw a link between the center-bottom of each node to the center-top of its child node(s).

Many enhancements can be made to this rather basic algorithm in order to improve space utilization and move child nodes closer to their parents.

Some of these include:

- a Rather than using even spacing and centering, divide each level based on the number of terminal nodes belonging to each subtree.
- Spread terminal nodes evenly across the drawing area and center parent nodes above them.
- Add some buffer space between adjacent nonsibling nodes to emphasize relationships.
- If possible, reorder the subtrees of a node to achieve more symmetry and balance.
- Position the root node in the center of the display and lay out child nodes radially, rather than vertically.

For large trees, a popular approach is to use the third dimension, supplemented with tools for rotation, translation, and zooming. Perhaps the most well-known of such techniques is called a *cone tree* [293]. In this layout, the children of a node are arranged radially at evenly spaced angles and then offset perpendicular to the plane. The two parameters critical to this process are the radius and offset distance; varying these influences the density of the display and the level of occlusion. Minimally they should be set so that



**Figure 8.6.**

All example of a hierarchy displayed with a cone tree [293]. (Image © 1991 Association of Computing Machinery. Reprinted by permission, courtesy of PARC, Inc.)

separate branches of the tree do not fall into the same section of 3D space. One method to ensure this is to have the radius inversely proportional to the depth of a node in the tree. In this manner, nodes close to the root are significantly separated, and those near the bottom of the tree are closer together. An example is shown in Figure 8.6.

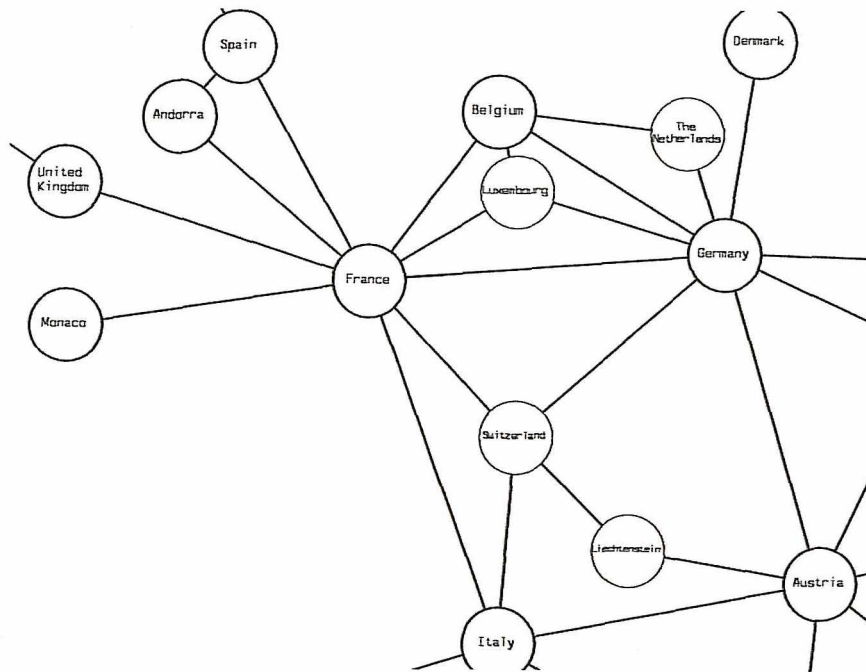
## 8.2 Displaying Arbitrary Graphs/Networks

Trees are just one type of a more general representation of relations called a graph. Technically speaking, a tree is a connected, unweighted, acyclic graph. Clearly, there are many other possibilities, including graphs with weighted edges, undirected graphs, graphs with cycles, disconnected graphs, and so on. Rather than give more algorithms specific to other classes of graphs, which could certainly fill more than a textbook, we will describe some general approaches for visualizing graphs in which the class or structure is not known, which we term an arbitrary graph. For our purposes, we will assume that the graph is undirected, though some of the techniques presented are easily extended to directed graphs. We will look at two distinct graph drawing approaches: node-link dagrams (building on the material from the previous section) and matrix displays. Readers interested in a broader or deeper exposure to graph drawing are directed to the vast amount of literature on this topic, some of which is listed at the end of the chapter.

### 8.2.1 Node-Link Graphs

Force-directed graph drawing methods use a spring analogy to represent the links, with node positions iteratively refined until the overall energy or stress of the system is minimized (see Figure 8.7). For each pair of connected nodes, there are two forces:  $f_{ij}$ , the force caused by the spring between them, and  $g_{ij}$ , an electrical repulsion force to keep nodes from getting too close. A simple model is to use Hooke's law to represent the spring force and an inverse square law to represent the repulsion force. If  $d(i, j)$  is the Euclidean distance between nodes  $i$  and  $j$ ,  $s_{i,j}$  is the natural spring length (at rest), and  $k_{ij}$  is the spring tension, the x-component of the spring force between two nodes can be computed as

$$f_{ij}(x) = k_{ij} * (d(i, j) - s_{ij}) * (x_i - x_j) / d(i, j).$$



**Figure 8.7.** An example of a force-directed graph. The graph, showing relationships between countries of Europe, was generated with aiSee: <http://www.aisee.com>.

If  $r_{ij}$  is the strength of the repulsion between nodes  $i$  and  $j$ , the  $x$ -component of the repulsion force can be computed as

$$g_{ij}(x) = (r_{ij}/d(i, j)^2) * (x_i - x_j)/d(i, j).$$

Thus, one step of the position refinement process would calculate the sum of all the forces on each node ( $x$ -,  $y$ -, and  $z$ -components, as appropriate) and move its position proportional to that force. Clearly, once points have moved, all the forces need to be recalculated and another shift of positions made. To avoid oscillation, it is common to start with movements that are a significant percentage of the force and then use smaller and smaller step sizes to converge on the point where the forces are minimized. Initial positions can be assigned randomly. As it is quite possible to end up in a local, rather than a global, energy minimum, it is common to run the layout algorithm multiple times with different initial configurations to find the best of several computed configurations. The goodness of the layout can be computed based on the sum of the magnitude of forces on a given configuration.

Planar graph drawing techniques start with the assumption that the underlying graph is planar, e.g., it has no edge crossings. These algorithms have gotten a lot of attention, for several reasons. First, as the theory of planar graphs has a long history, there are many concepts that can be exploited from the literature. Second, as edge-crossings tend to make graphs difficult to read, it is a good strategy to minimize or eliminate such crossings. Finally, planar graphs tend to be sparse; Euler's formula indicates that a planar graph with  $n$  vertices has at most  $3n - 6$  edges. Concentrating on planar graphs is not overly restrictive, as one can eliminate crossings by inserting dummy nodes at the crossings, perform the layout using a planar graph algorithm, and then remove the dummy nodes.

We will, in addition, assume that the graph is *connected*, e.g., there is a path from every node to every other node. Graphs that are not connected can be separated into subgraphs that can be drawn separately. A subgraph that is maximally connected (all nodes are connected) is a connected component of the graph. Other useful definitions include:

- A face is a partition of the plane isolated by a set of connected vertices.
- A neighbor set is a counter-clockwise listing of the vertices incident to a particular vertex.
- A planar embedding is a class of planar graph drawings with the same neighbor sets for each vertex. A planar graph can have an exponential number of such embeddings.
- A cutvertex is any node that causes the graph to be disconnected if it is removed.
- A biconnected graph is one without a cutvertex.
- m A block is a maximally biconnected subgraph of a graph.
- m A separating pair means two vertices whose removal causes a biconnected graph to become disconnected.
- A triconnected graph is one without a separating pair. A planar triconnected graph has a unique embedding.

We first need a strategy for determining if a graph is planar. Several such algorithms exist, though efficient ones have a very high degree of complexity and simple ones tend to be computationally expensive. We can start by simplifying the problem a bit. We do this by noting that a graph is planar



only if all of its connected components are also planar. Similarly, we can state that a connected graph is planar only if all its biconnected components are planar. Thus, we just need an algorithm that determines if a biconnected graph is planar or not.

The general reasoning of the algorithm is as follows. We will perform a divide-and-conquer approach by noting that if our graph contains a cycle such that no other cycle is present that doesn't contain an edge of the original cycle (e.g., there aren't cycles left when the edges involved in the original cycle are removed), what remain are paths that start and stop on one of the vertices of the cycle (called *attachments*). These *pieces* of the graph can be drawn either within the cycle or outside the cycle. Two such pieces *interlace* if they both start and end on nodes of the cycle, and the two ends of one piece are separated by one end of the other piece. To be drawn in a planar fashion, one of these interlaced pieces would need to be drawn inside the cycle, and the other on the outside. If we now create a graph of all the pieces, with an edge between two pieces if they interlace, as long as this graph is *bzpartite* (separable into two sets of vertices such that no edge exists between members of the same set), the original graph is planar. Figure 8.8 shows examples of these components. Note that there are a couple of instances of interlacing among the parts.

If the graph contains more cycles after removing the edges of the original cycle, this means that one or more of the pieces contains a cycle (see the purple piece in Figure 8.8). In this case, we create a subgraph containing this piece and a section of the original cycle connecting the end points of the part, and recursively call the planarity test algorithm. The pseudocode

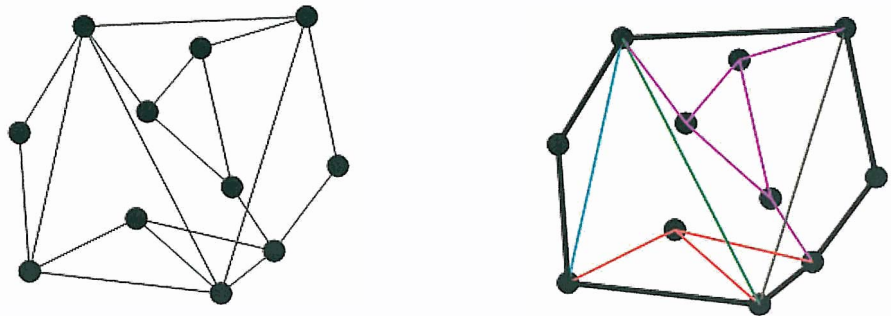


Figure 8.8.

An example of a biconnected graph, a cycle (in black), and the five pieces (in different colors).



for this algorithm is as follows [20]. Note that a *separating cycle* is one that generates at least two pieces.

Given a biconnected graph  $G$  and a separating cycle  $C$ :

1. Compute all the pieces of  $G$  with respect to  $C$ .
2. For each piece  $P$  that is not a simple path (e.g., that contains a cycle).
  - (a) Create graph  $G'$  consisting of  $P$  plus  $C$ .
  - (b) Create cycle  $C'$  consisting of a path through  $P$  plus the section of  $C$  joining the ends.
  - (c) Apply the algorithm to  $(G', C')$ . If the result is nonplanar,  $G$  is nonplanar.
3. Compute the interlacement graph  $I$  of the pieces of  $G$ .
4. If  $I$  is not bipartite,  $G$  is nonplanar; else  $G$  is planar.

If a graph is nonplanar, we can make it planar using the following strategy:

1. Determine the largest planar subgraph of the graph.
2. For the remaining vertices, place each within a face that minimizes the number of edge crossings.
3. For each edge crossing, break the edges into two parts each, and connect the broken ends to a new dummy vertex.

Once a graph has been either determined to be planar or has been augmented to achieve planarity, there are many possible strategies for generating a drawing. One such technique, called the *visibility approach* [20], consists of a two-step process. In the first step, called the *visibility step*, a *visibility representation* of the graph is formed. In such a representation, each vertex is depicted as a horizontal line segment, and each edge is depicted as a vertical line connecting the corresponding vertex segments. It should be clear that for a planar graph, it is always possible to draw such a representation without crossing edges other than where they meet the vertex segments. Obviously, many possible orderings of the vertex segments are possible; one strategy would be to arrange them to minimize the total length of the vertical connectors.

In the second step, called the *replacement step*, each vertex segment is collapsed to a single point, and each vertical connector is replaced by a

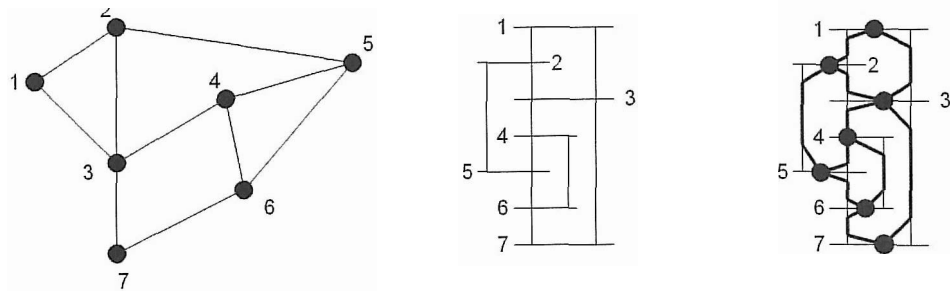


Figure 8.9. The stages of drawing a planar graph. From left to right: original graph, visibility representation, and replacement step.

polyline that follows the original edge as much as possible, with a segment at each end connecting the edge to its corresponding vertex. Many options exist for the replacement step, including the location of the nodes and the strategy used to form the connections (e.g., straight versus curved lines, single segment versus multiple segments). An example of the process is shown in Figure 8.9.

### 8.2.2 Matrix Representations for Graphs

An alternate visual representation of a graph is via an adjacency *matrix*, which is an  $N$  by  $N$  grid (where  $N$  is the number of nodes), where position  $(i, j)$  represents the existence (or not) of a link between nodes  $i$  and  $j$ . This may be a binary matrix, or the value might represent the strength or weight of the link between the two nodes. This method overcomes one of the biggest problems with node-link diagrams, namely that of crossing edges, though it doesn't scale well to graphs with large numbers (thousands) of nodes. Bertin [26] was one of the first researchers to investigate the power of this representation, using different reordering strategies to organize the rows and columns to reveal structures within the graph. The importance of the reordering is apparent in Figure 8.10, where each matrix represents the same eight-node graph. The two four-node cliques are clearly apparent in the second display.

There have been numerous algorithms proposed for reordering the rows and columns of the matrix to expose the most structure. Some are primarily user-driven, which would support ordering based on the values in one of the rows or columns as a starting point. Others are purely automatic, which

	a	b	c	d	e	f	g	h
a		•	•			•		
b	•			•		•		
c	•				•		•	•
d	•	•				•		
e			•				•	•
f	•	•		•				
g			•		•			•
h			•		•		•	

	p	q	r	s	t	u	v	w
p		•	•	•				
q	•		•	•				
r	•	•		•				
s	•	•	•		•			
t				•		•	•	•
u					•		•	•
v					•	•		•
w					•	•	•	

Figure 8.10. Two matrix displays of the same graph, using different orderings of nodes. Structure is more clearly present in the matrix on the right.

rely on some metric for evaluating a particular ordering and a strategy for generating orders to test. As in any optimization process, there is a good chance that finding the optimal ordering is NP-complete (namely, that no algorithm of polynomial or less complexity can be found). Thus, a number of heuristics have been proposed over the years that generally result in good orderings, especially for certain classes of graphs.

As an example, we can use a simplistic order evaluation strategy, namely to count the number of occurrences of matching elements in adjacent rows or columns. This tends to group nodes that link or don't link to a common node. In Figure 8.10, the left-most matrix has a score of 9 when counting only vertical neighbors, while the right-most matrix has a score of 20. By enumerating all possible orders, we can find the orderings that give the highest match score. For modest numbers of nodes, this would be an acceptable strategy, but since the number of possible orderings is on the order of  $N!$ , this approach does not scale well. Ordering of nodes is similar to the traveling salesman problem (TSP), where one tries to find a path that passes through a collection of cities without visiting any city more than once, while at the same time minimizing the total distance traveled. As this is basically the same problem as finding the ordering of the rows or columns of a matrix to minimize some metric, heuristic solutions that have been used for the TSP can also be employed here.

### 8.3 Other Issues

Once a basic visualization of a tree or graph has been developed, there are a number of additional considerations, primarily addressing the issue of interpretability. Two such important considerations will be elaborated upon in this section: labeling and interaction.

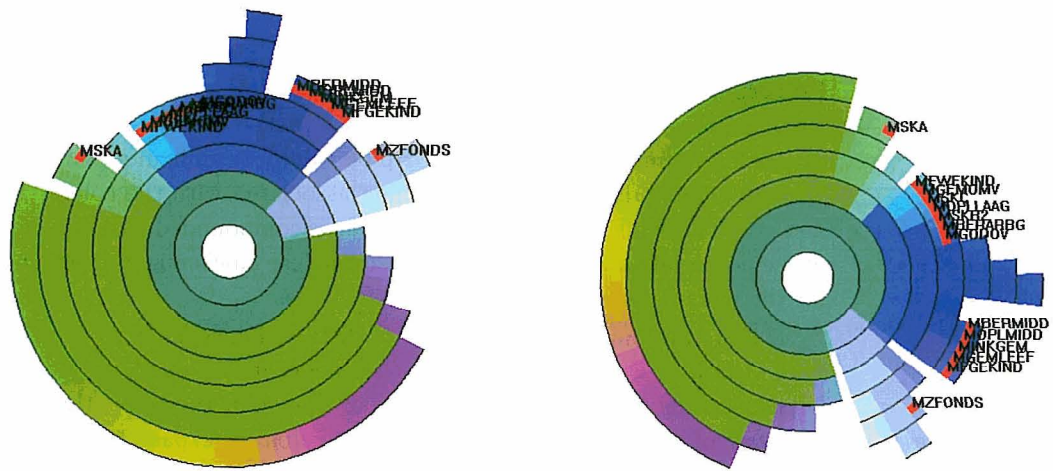
### 8.3.1 Labeling

Proper labeling of a visualization is crucial to allow a viewer to understand what is being shown. A map would be of little value without some form of labeling; similarly, a color-coded plot would be difficult to understand without some indication of the meaning associated with the colors. In tree and graph drawing, the problem of labeling is compounded, not only because of the potential for many nodes, but also because labels might also be needed for the links between nodes.

If there are only a small number of distinct labels, such as showing the type of link or a class associated with a node, it is best to use nontextual labels, such as the color, size, or shape of a node or the color, thickness, or line style of a link. This does not require much screen space and can usually be interpreted unambiguously even in the presence of a modest amount of line crossing and node occlusion. However, if the number of distinct labels exceeds five or six, the likelihood of misinterpretation can become large. A key for interpreting the graphical attribute mapping is essential.

For small graphs, a common strategy for node labeling is to put the labels within the nodes, using rectangular or oval node shapes to accommodate the text. To avoid distorting the perception of the nodes, the size of the nodes should be dictated by the length of the longest label. For situations where the labels can be very long, one option is to use abbreviations or numeric labels, along with a key for interpretation. Viewers will eventually learn the correspondences between the shortened labels and their actual meaning. A similar strategy can be used for edge labeling, placing the labels near the center of the edge. For edges that are predominantly vertical, these should be to the left or right of the edge, while for predominantly horizontal edges, they should be above or below. Using a consistent strategy will reduce the potential for erroneously associating a label with the wrong edge.

At the other extreme, if there are a large number of distinct labels that need to be shown, or the labels themselves are quite long, it becomes readily apparent that simultaneous display of all labels will be ineffective. Several strategies have been developed to cope with this problem. A common solution is to only show labels in a small region of the graph, for example, within a certain radius of the cursor position. If the density of the display is too high, a distortion of the visualization may be required (see the next subsection) to provide more screen space for that section of the graph. All alternate to distortion that sometimes works is to rotate the graph to reduce the overlap between labels (see Figure 8.11). Another interesting solution proposed in [37] is to only show a random subset of the labels for a short



**Figure 8.11.** Improving the readability of labels via rotation. (Image from [412], © 2003 Palgrave Macmillan.)

period of time, and then switch to showing the labels for a different subset. The idea behind this approach is that the viewer's short-term memory will enable recall of a larger number of labels as compared to a static display, especially if this memory is refreshed on a regular basis.

### 8.3.2 Interactions

Even though Chapters 10 and 11 of this text are dedicated to interactions within visualization environments, there are a few interaction techniques that are most relevant to tree and graph visualization that will be described in this chapter. Some types of interaction, such as panning and zooming, are common to all types of visualization, and thus will only be briefly mentioned here for completeness. Others, such as focus+context, while applicable to a wide range of visualizations, have been primarily developed in the area of tree and graph visualization and will thus be described in more detail here.

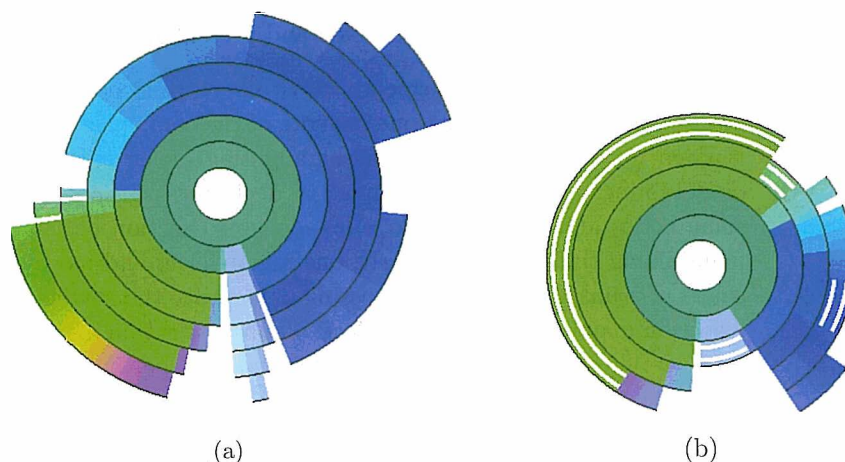
**Interactions with the virtual camera.** Interactions such as panning, zooming, and rotation can be viewed as simple changes to the virtual camera being used to capture a segment of a scene. These allow the viewer to incrementally build up a mental model of the objects of the scene and their interrelationships. Operations of this type are often manually controlled, though

automated techniques such as data-driven fly-throughs and spinning of 3D objects can be automatically derived and presented.

Interactions with the graph elements. Most interactions of this type start with a *selection* operation, where one or more of the components of the graph are isolated for some action, such as highlighting, deleting, masking, moving, or obtaining details. For example, to declutter a graph one might select some nodes and drag them to a less-occupied section of the screen, while maintaining their links. Similarly, one might select and move or change the shape of a link to eliminate a crossing or improve the aesthetics of a graph. Selection may involve a single object, all objects within a specified region or distance, or a set of objects that satisfy a user-specified set of constraints (e.g., all nodes directly connected to a given node). One of the biggest problems with selecting elements in a graph occurs in dense regions of the drawing, where elements are so close together that unambiguous selection is difficult or impossible. This exposes the need for other types of interaction, such as zooming or the distortion techniques described later.

Interactions with the graph structure. There are two classes of interactions that are directed at the graph structure. The first class result in changes to the structure itself. For example, reordering the branches of a tree may expose relationships that were not apparent in the original ordering. Redrawing a graph with different weights on the constraints can generate graphs that make certain tasks easier to perform. Reordering the columns or rows in a matrix visualization can expose new features or relations within the data. Techniques within this class are often very specific to the type of graph being shown.

A second class of interactions associated with the graph structure comprises the so-called *focus+context* techniques, where a selected subset of the structure (focus) is presented in detail, while the rest of the structure is shown in low detail to help the viewer maintain context. These techniques are related to panning and zooming, without the loss of context. The most popular of these distortion techniques are the many variants on a fisheye lens, where the parts of the visualization falling within a focal region are enlarged using a nonlinear scaling, while the parts outside the focal region are proportionally shrunk to maintain their presence in the display. This distortion can be performed either in *screen space* (i.e., based on pixels) or in *structure space* (i.e., based on the components of the graph). It is the latter case that is more interesting in graph visualization, as we might, for example, enlarge one branch of a tree while reducing the size of other branches, or enlarge all links within three connections of a particular node in order to view



**Figure 8.12.** Some interaction operations on sunburst displays: (a) the blue subtree has been expanded, while the rest of the tree has been compressed; (b) several subtrees have been rolled up to simplify the display. (Image from [412], © 2003 Palgrave Macmillan.)

its neighborhood in more detail. An example of structure space distortion can be seen in Figure 8.12, where the blue subtree of Figure 8.11 has been angularly enlarged to enable easier exploration and interactive selection.

A technique that can be considered related to both of these classes is that of selective hiding or removal of sections of the graph. For example, once a branch of a tree has been thoroughly investigated, the user might want to remove it from the display to provide more space for the unexplored regions. In a sense, this can be seen as changing the structure (deleting a component), or as reducing the level of detail for the branch to its root. The terms *roll-up* and *drill-down* are often used to describe the process of hiding and exposing details in a visualization. Figure 8.12 shows several subtrees that have been rolled up, with the double white band informing the user that details exist under those nodes.

## 8.4 Related Readings

Robertson et al. [293] and Brian Johnson and Ben Schneiderman [176] introduce the concepts of cone trees and treemaps, respectively. John Stasko and Eugene Zhang [336] describe one of several variants on radial space-

filling techniques for tree visualization. The book, *Graph Drawing: Algorithms for the Visualization of Graphs* [20] is an excellent introduction to the field of graph drawing. *The Semiology of Graphs* [26] by J. Bertin is the seminal work on reorderable matrix representations for graphs. Herman et al. [159] presents a survey of graph visualization and interactions with graphs. The paper by Leung and Apperley [232] contains a comprehensive survey of distortion techniques, many of which are applicable to tree and graph visualizations.

## 8.5 Exercises

1. Give some examples of how rules for graph drawing can conflict with each other.
2. Compare rectilinear and radial space-filling tree visualization techniques. Under what conditions, or for what tasks, is one better or worse than the other?
3. Compare node-link and matrix graph visualization techniques. Under what conditions, or for what tasks, is one better or worse than the other?
4. What is the smallest node-link graph (e.g., smallest number of nodes and links) that you can devise that is nonplanar?

## 8.6 Projects

1. Write a program that reads in a graph in the following format:

```
number-of-vertices number-of-edges
edge1_start edge1_end
edge2_start edge2_end
...
edgeN_start edgeN_end
```

Add a very simple drawing function that places the vertices in random positions and connects the vertices based on the edge list. Run the program several times with a data set of your design (it should have more than 10 nodes and 20 edges). What conclusions can you draw from observing the output?



2. Modify the above program to place the vertices at equal angles around a circle. Again, run the program several times and describe your observations. From these observations, can you propose a vertex-ordering algorithm that will generally result in less cluttered displays?
3. Write a program that will determine if a graph entered in the above format is connected, e.g., if there is a path from every node to every other node.
4. Write a program that will determine if a graph entered in the above format is biconnected, e.g., if removal of a single node will not disconnect the graph.
5. Assuming that the input graph represents a tree, and that all links are given in the order of (parent, child), write a program that will draw the tree as in Figure 8.5, where all nodes on the same level are evenly spaced. (Hint: in a single pass through the list of links, you should be able to assign each node to a level.)
6. Modify the above program to generate a *radial* layout, e.g., the layers are arranged as concentric circles with a radius proportional to the tree depth.
7. Modify either or both of the above programs to insert extra space between adjacent nonsibling nodes.
8. Write a program that generates the adjacency matrix  $A$  using the same data as in Project 1 or some other graph data. Use R-project (or your own code) to compute  $A^2$  and draw it differentiating the values in the matrix using color (note that it may have values larger than 1). Explain what you see and the meaning of the numbers.