Diagrama de árbol

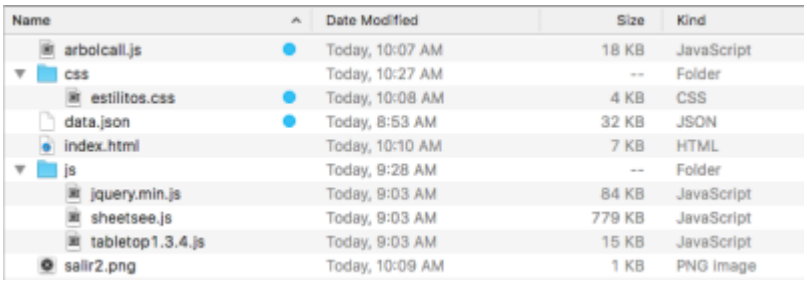
Esta gráfica se genera usando D3.js, específicamente la API que concierne a estructuras de árbol. API tree layaout

Uso: Es un tipo de herramienta que sirve para determinar todos los posibles cursos, resultados o acontecimiento de un experimento, en función de un nodo padre y un nodo hijo.

Archivos base:

arbol.zip

Estructura de los archivos base.



Name	Date Modified	Size	Kind
arbolcall.js	Today, 10:07 AM	18 KB	JavaScript
css	Today, 10:27 AM	--	Folder
estilitos.css	Today, 10:08 AM	4 KB	CSS
data.json	Today, 8:53 AM	32 KB	JSON
index.html	Today, 10:10 AM	7 KB	HTML
js	Today, 9:28 AM	--	Folder
jquery.min.js	Today, 9:03 AM	84 KB	JavaScript
sheetsee.js	Today, 9:03 AM	779 KB	JavaScript
tabletop1.3.4.js	Today, 9:03 AM	15 KB	JavaScript
salir2.png	Today, 10:09 AM	1 KB	PNG image

Los que tienen el punto azul con los archivos que se pueden modificar. Los otros archivos no se deberían tocar.

PASO A PASO

La data: El formato con el cual trabaja este gráfica es.json. Debido a la naturaleza de este tipo de gráfico, la estructura de la data será de este tipo.

Por lo general, siempre la data se trabaja en Excel o en google spreadsheets. Se recomienda trabajar la data en spreadsheets porque al spreadsheets se le puede adicionar un script para facilitar la exportación de la data en json, lo explicaré más adelante.

La estructura de datos en spreadsheets

A	B	C	D	E
ID	name	parent	value	type
1	CPD	null	15	black
2	E- Programación	CPD	8	lightgray
3	Contacto inicial (1)	E- Programación	4	mediumseagreen
4	Captura de información del cliente (1)	Contacto inicial (1)	4	mediumseagreen
5	No ha sido creado como cliente	Captura de información del cliente (1)	2	darkorange
6	No está creada la obra	Captura de información del cliente (1)	2	darkorange
7	No está activada la obra	Captura de información del cliente (1)	2	silver
8	No es clara la dirección	Captura de información del cliente (1)	2	silver
9	Persona no autorizada	Captura de información del cliente (1)	2	silver
10	Ya se realizó programación	Captura de información del cliente (1)	2	silver
11	Captura de información del pedido (1)	Captura de información del cliente (1)	4	mediumseagreen
12	Desconocimiento técnico	Captura de información del pedido (1)	2	silver
13	Confirma disponibilidad	Captura de información del pedido (1)	4	mediumseagreen
14	No hay disponibilidad	Confirma disponibilidad	2	silver
15	Pedido retenido por cartera	Confirma disponibilidad	2	darkorange

En la tabla, las columnas señaladas con **color**, son aquellas columnas imprescindibles para poder generar el json:

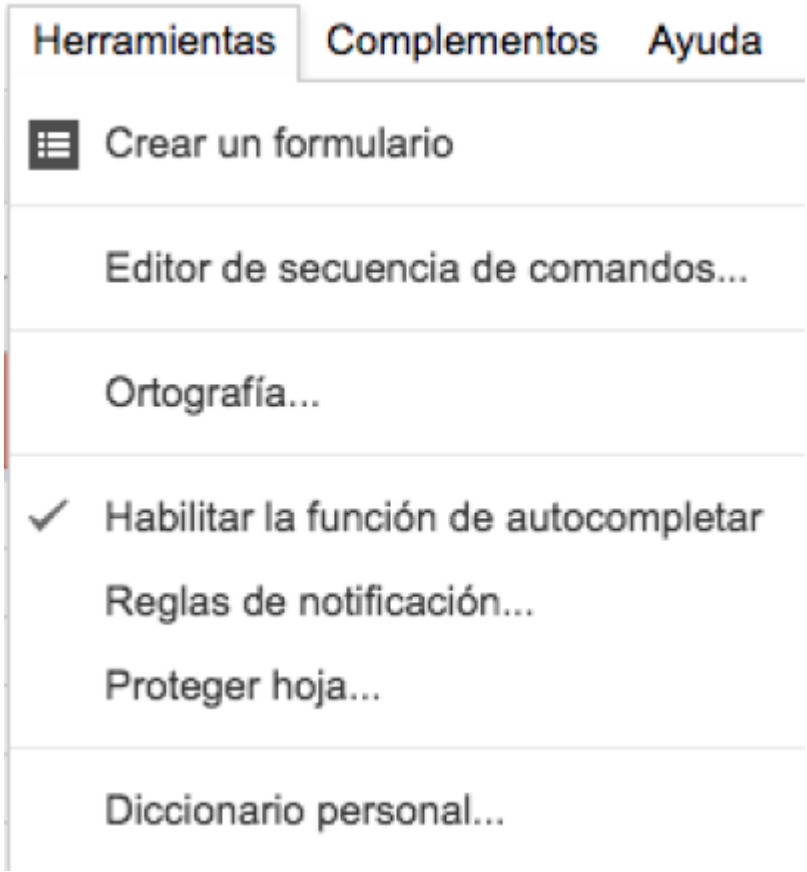
- **Id:** Debe ser Int (Un número no decimal)
- **name:** Es un string (cadena de texto) y por lo general son los nombres de los nodos
- **parent:** Es un string (cadena de texto) y aquí van los parientes de los nodos names.
- **type:** Es el color de los nodos y por consiguiente su link (línea conectora entre name y parent)

El campo value no afecta a la gráfica, pero tampoco se puede eliminar. Por default value debe ser 1.

¿ Cómo generar el json ?

Para generar el json, se debe hacer lo siguiente.

Escoger una celda vacía del spritesheet y luego ir al menú Herramientas → Editor de secuencia de comandos.



Allí se nos abrirá un nuevo documento donde podemos escribir el código para la tabla, entonces en este archivo en blanco; copiamos y pegamos el siguiente script.

<script lang="javascript"> *Includes functions for exporting active sheet or all sheets as JSON object (also Python object syntax compatible).* Tweak the makePrettyJSON_ function to customize what kind of JSON to export.

var FORMAT_ONELINE = 'One-line'; var FORMAT_MULTILINE = 'Multi-line'; var FORMAT_PRETTY = 'Pretty';

var LANGUAGE_JS = 'JavaScript'; var LANGUAGE_PYTHON = 'Python';

var STRUCTURE_LIST = 'List'; var STRUCTURE_HASH = 'Hash (keyed by "id" column)';

/* Defaults for this particular spreadsheet, change as desired */ var DEFAULT_FORMAT = FORMAT_PRETTY; var DEFAULT_LANGUAGE = LANGUAGE_JS; var DEFAULT_STRUCTURE = STRUCTURE_LIST;

function onOpen() {

```
var ss = SpreadsheetApp.getActiveSpreadsheet();
var menuEntries = [
  {name: "Export JSON for this sheet", functionName: "exportSheet"},
  {name: "Export JSON for all sheets", functionName: "exportAllSheets"},
  {name: "Configure export", functionName: "exportOptions"},
];
ss.addMenu("Export JSON", menuEntries);
```

}

function exportOptions() {

```
var doc = SpreadsheetApp.getActiveSpreadsheet();
var app = UiApp.createApplication().setTitle('Export JSON');

var grid = app.createGrid(4, 2);
grid.setWidget(0, 0, makeLabel(app, 'Language:'));
grid.setWidget(0, 1, makeListBox(app, 'language', [LANGUAGE_JS,
LANGUAGE_PYTHON]));
grid.setWidget(1, 0, makeLabel(app, 'Format:'));
grid.setWidget(1, 1, makeListBox(app, 'format', [FORMAT_PRETTY,
FORMAT_MULTILINE, FORMAT_ONELINE]));
grid.setWidget(2, 0, makeLabel(app, 'Structure:'));
grid.setWidget(2, 1, makeListBox(app, 'structure', [STRUCTURE_LIST,
STRUCTURE_HASH]));
grid.setWidget(3, 0, makeButton(app, grid, 'Export Active Sheet',
'exportSheet'));
grid.setWidget(3, 1, makeButton(app, grid, 'Export All Sheets',
'exportAllSheets'));
app.add(grid);

doc.show(app);
```

}

function makeLabel(app, text, id) {

```
var lb = app.createLabel(text);
if (id) lb.setId(id);
return lb;
```

}

function makeListBox(app, name, items) {

```
var listBox = app.createListBox().setId(name).setName(name);
listBox.setVisibleItemCount(1);

var cache = CacheService.getPublicCache();
var selectedValue = cache.get(name);
Logger.log(selectedValue);
for (var i = 0; i < items.length; i++) {
  listBox.addItem(items[i]);
  if (items[1] == selectedValue) {
    listBox.setSelectedIndex(i);
  }
}
return listBox;
```

}

function makeButton(app, parent, name, callback) {

```
var button = app.createButton(name);
app.add(button);
var handler =
app.createServerClickHandler(callback).addCallbackElement(parent);;
button.addClickHandler(handler);
return button;
```

}

function makeTextBox(app, name) {

```
var textArea    =
app.createTextArea().setWidth('100%').setHeight('200px').setId(name).setName
(name);
return textArea;
```

}

function exportAllSheets(e) {

```
var ss = SpreadsheetApp.getActiveSpreadsheet();
var sheets = ss.getSheets();
```

```
var sheetsData = {};
for (var i = 0; i < sheets.length; i++) {
  var sheet = sheets[i];
  var rowsData = getRowsData_(sheet, getExportOptions(e));
  var sheetName = sheet.getName();
  sheetsData[sheetName] = rowsData;
}
var json = makeJSON_(sheetsData, getExportOptions(e));
return displayText_(json);
```

}

function exportSheet(e) {

```
var ss = SpreadsheetApp.getActiveSpreadsheet();
var sheet = ss.getActiveSheet();
var rowsData = getRowsData_(sheet, getExportOptions(e));
var json = makeJSON_(rowsData, getExportOptions(e));
return displayText_(json);
```

}

function getExportOptions(e) {

```
var options = {};

options.language = e && e.parameter.language || DEFAULT_LANGUAGE;
options.format   = e && e.parameter.format || DEFAULT_FORMAT;
options.structure = e && e.parameter.structure || DEFAULT_STRUCTURE;

var cache = CacheService.getPublicCache();
cache.put('language', options.language);
cache.put('format',   options.format);
cache.put('structure',   options.structure);

Logger.log(options);
return options;
```

}

function makeJSON_(object, options) {

```
if (options.format == FORMAT_PRETTY) {
  var jsonString = JSON.stringify(object, null, 4);
} else if (options.format == FORMAT_MULTILINE) {
  var jsonString = Utilities.jsonStringify(object);
  jsonString = jsonString.replace(/},/gi, '},\n');
  jsonString = prettyJSON.replace(/":\[{"/gi, '":\n[{"');
  jsonString = prettyJSON.replace(/}\],/gi, '}],\n');
} else {
  var jsonString = Utilities.jsonStringify(object);
```

```
}
if (options.language == LANGUAGE_PYTHON) {
  // add unicode markers
  jsonString = jsonString.replace(/"([a-zA-Z]*)":\s+"/gi, '"$1": u"');
}
return jsonString;
```

}

function displayText_(text) {

```
var app = UiApp.createApplication().setTitle('Exported JSON');
app.add(makeTextBox(app, 'json'));
app.getElementById('json').setText(text);
var ss = SpreadsheetApp.getActiveSpreadsheet();
ss.show(app);
return app;
```

}

getRowsData iterates row by row in the input range and returns an array of objects. Each object contains all the data for a given row, indexed by its normalized column name. *Arguments:* - sheet: the sheet object that contains the data to be processed - *range: the exact range of cells where the data is stored* - columnHeadersRowIndex: specifies the row number where the column names are stored. *This argument is optional and it defaults to the row immediately above range;* Returns an Array of objects. function getRowsData_(sheet, options) {

```
var headersRange = sheet.getRange(1, 1, sheet.getFrozenRows(),
sheet.getMaxColumns());
var headers = headersRange.getValues()[0];
var dataRange = sheet.getRange(sheet.getFrozenRows()+1, 1,
sheet.getMaxRows(), sheet.getMaxColumns());
var objects = getObjects_(dataRange.getValues(),
normalizeHeaders_(headers));
if (options.structure == STRUCTURE_HASH) {
  var objectsById = {};
  objects.forEach(function(object) {
    objectsById[object.id] = object;
  });
  return objectsById;
} else {
  return objects;
}
```

}

getColumnsData iterates column by column in the input range and returns an array of objects. Each object contains all the data for a given column, indexed by its normalized row name. *Arguments:* - sheet: the sheet object that contains the data to be processed - *range: the exact range of cells where the data is stored* - rowHeadersColumnIndex: specifies the column number where the row names are stored. *This argument is optional and it defaults to the column immediately left of the range;* Returns

an Array of objects. function getColumnsData_(sheet, range, rowHeadersColumnIndex) {

```
rowHeadersColumnIndex = rowHeadersColumnIndex || range.getColumnIndex() - 1;
var headersTmp = sheet.getRange(range.getRow(), rowHeadersColumnIndex,
range.getNumRows(), 1).getValues();
var headers = normalizeHeaders_(arrayTranspose_(headersTmp)[0]);
return getObjects(arrayTranspose_(range.getValues()), headers);
```

}

For every row of data in data, generates an object that contains the data. Names of object fields are defined in keys. *Arguments:* - data: JavaScript 2d array - *keys: Array of Strings that define the property names for the objects to create function getObjects_(data, keys) { var objects = []; for (var i = 0; i < data.length; ++i) { var object = {}; var hasData = false; for (var j = 0; j < data[i].length; ++j) { var cellData = data[i][j]; if (isCellEmpty_(cellData)) { continue; } object[keys[j]] = cellData; hasData = true; } if (hasData) { objects.push(object); } } return objects; }* Returns an Array of normalized Strings. *Arguments:* - headers: Array of Strings to normalize function normalizeHeaders_(headers) {

```
var keys = [];
for (var i = 0; i < headers.length; ++i) {
  var key = normalizeHeader_(headers[i]);
  if (key.length > 0) {
    keys.push(key);
  }
}
return keys;
```

}

Normalizes a string, by removing all alphanumeric characters and using mixed case to separate words. The output will always start with a lower case letter. *This function is designed to produce JavaScript object property names.* Arguments: - *header: string to normalize* Examples: *“First Name”* → *“firstName”* “Market Cap (millions) → “marketCapMillions *“1 number at the beginning is ignored”* → *“numberAtTheBeginningIsIgnored” function normalizeHeader_(header) { var key = ””; var upperCase = false; for (var i = 0; i < header.length; ++i) { var letter = header[i]; if (letter == “ ” && key.length > 0) { upperCase = true; continue; } if (!isAlnum_(letter)) { continue; } if (key.length == 0 && isDigit_(letter)) { continue;* first character must be a letter

```
  }
  if (upperCase) {
    upperCase = false;
    key += letter.toUpperCase();
  } else {
    key += letter.toLowerCase();
  }
}
return key;
```

}

Returns true if the cell where cellData was read from is empty. Arguments: - *cellData: string function isCellEmpty_(cellData) { return typeof(cellData) == "string" && cellData == ""; }* Returns true if the character char is alphabetical, false otherwise. function isAlnum_(char) {

```
return char >= 'A' && char <= 'Z' ||
  char >= 'a' && char <= 'z' ||
  isDigit_(char);
```

}

Returns true if the character char is a digit, false otherwise. function isDigit_(char) { return char >= '0' && char <= '9'; } Given a JavaScript 2d Array, this function returns the transposed table. *Arguments:* - data: JavaScript 2d Array *Returns a JavaScript 2d Array* Example: arrayTranspose(1,2,3],[4,5,6) returns 1,4],[2,5],[3,6. function arrayTranspose_(data) {

```
if (data.length == 0 || data[0].length == 0) {
  return null;
}
```

```
var ret = [];
for (var i = 0; i < data[0].length; ++i) {
  ret.push([]);
}
```

```
for (var i = 0; i < data.length; ++i) {
  for (var j = 0; j < data[i].length; ++j) {
    ret[j][i] = data[i][j];
  }
}
```

```
return ret;
```

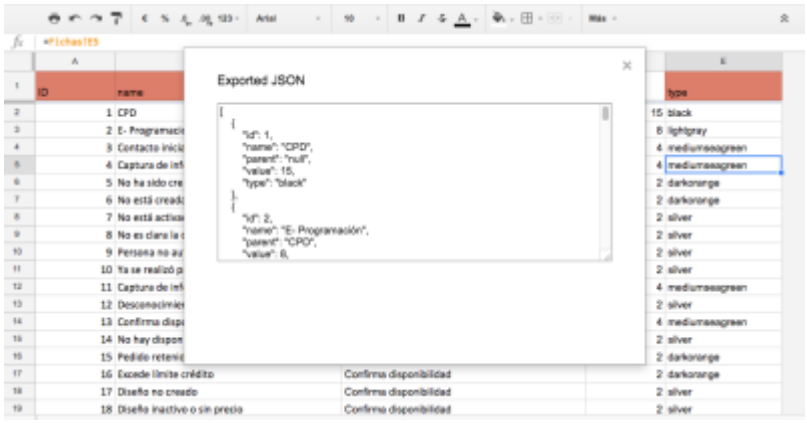
}

</script>

Finalizado la parte del script, guardamos y salimos. Volvemos a abrir el documento y nos aparecerá un nuevo botón donde podremos exportar la data en formato json, el nombre del botón es "Export JSON".

Para finalizar, hacemos click en este botón y le damos en la opción "export json for this sheet" que nos exportará la data en el formato json que necesitamos.

IMPORTANTE: Por defecto, el nombre de las columnas en la tabla debe llamarse igual a como están los nombres indicados y en ese mismo orden. (Id, name, parent, value, type).



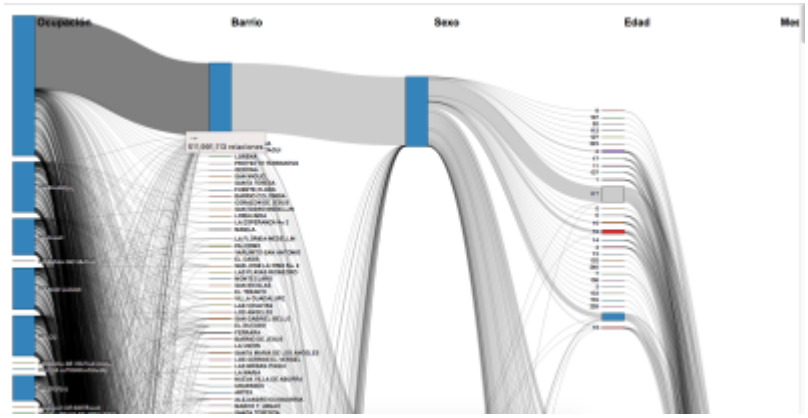
Insertar el gráfico en la web.

Para insertar la data en la gráfica, solo hace falta exportar el archivo con las indicaciones dadas anteriormente. Luego guardar el archivo en la carpeta /data. Se sugiere guardar el archivo con el nombre data.json; Pero si lo desea, puede guardar el archivo con un nombre diferente y renombrar la línea 6 del script para leer su archivo.

Con eso será suficiente para que la gráfica esté lista.

```
// ********** Convertimos la data en estructura de árbol ***************
d3.json("data/data.json", function(error, data) {
    var dataMap = data.reduce(function(map, node) {
        map[node.name] = node;
        return map;
    }, {});
```

Diagrama tipo sankey



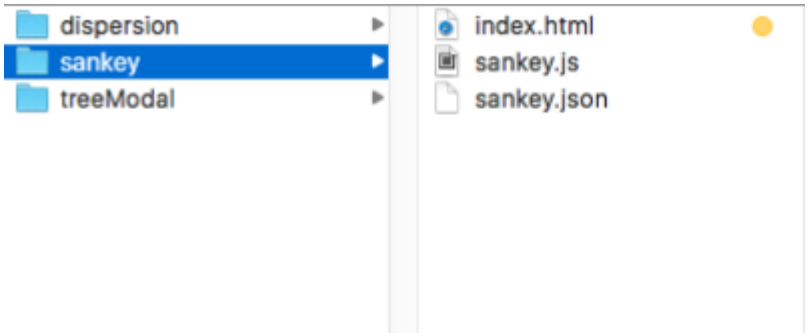
Esta gráfica se genera usando D3.js, específicamente la API que concierne a estructuras de tipo jerárquico. https://github.com/d3/d3/blob/master/API.md#hierarchies-d3-hierarchy

Uso: Este tipo de gráfico se caracteriza por presentar todas las relaciones posibles de una fuente o sector hacia varios sectores o un único sector, dependiendo de las relaciones que estén presentadas en la data.

Archivo base para descargar:

sankey.zip

Estructura de los archivos base



Para este gráfico solo es necesario modificar el archivo **index.hmtl.**

====== PASO A PASO

La data

El formato con el cual trabaja este gráfica es .json. Debido a la naturaleza de este tipo de gráfico, la estructura de la data será de este tipo.

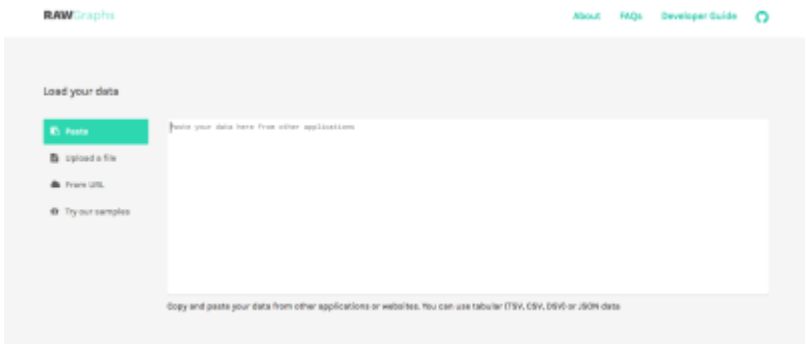
Por lo general, siempre la data se trabaja en Excel o en google spreadsheets. Para este gráfico la data se puede trabajar donde se sienta más cómodo.

La estructura de datos en spreadsheets o excel

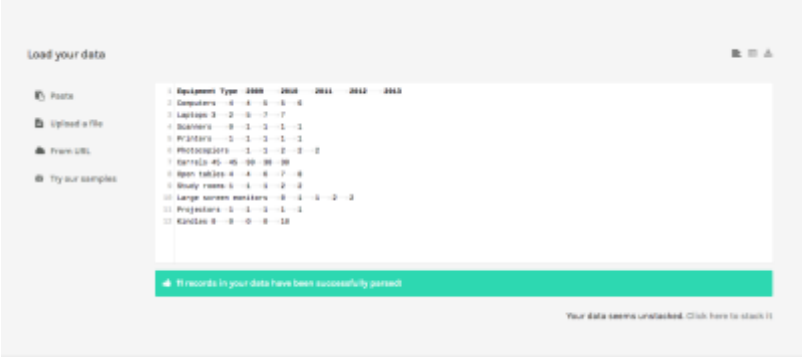
Los nombres de las columnas, y su contenido no afectan al código de este gráfico, es decir; no necesitamos nombres de columnas específicos para poder generar la data. Aunque podemos generar este gráfico con un archivo .csv, esto implica otro tratamiento de los datos que resulta más complicado de realizar, por tal razón; escogí un camino más fácil para que cualquier persona que lea esta guía pueda visualizar la data.

2 . ¿ Cómo se genera el json para este gráfico ?

EL primer paso es ir al siguiente enlace: http://app.rawgraphs.io/

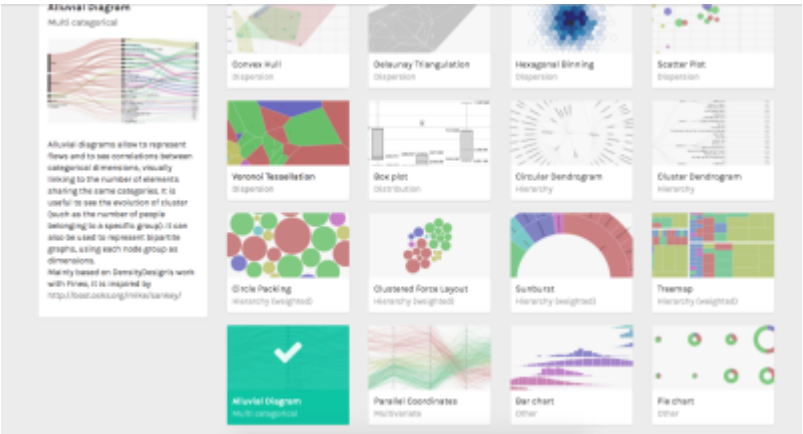


Aquí nos encontraremos con una zona blanca que nos indica que podemos cargar los datos mediante el botón verde “upload a file” o pegarlos directamente en la ventana.

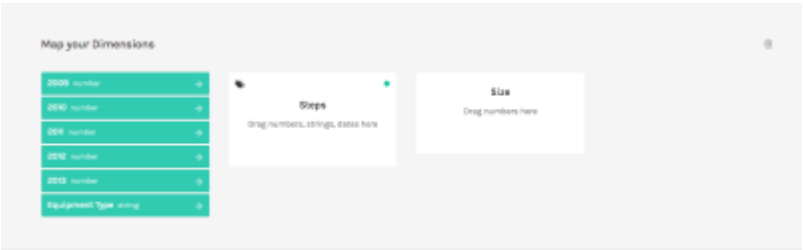


Cuando hayamos cargado o pegado los datos, la página nos indicará que estamos listos para ir al paso siguiente.

Nos dirigimos más abajo en la misma, y escogemos un gráfico, para este tutorial específico, tenemos que escoger “alluvial diagram”.

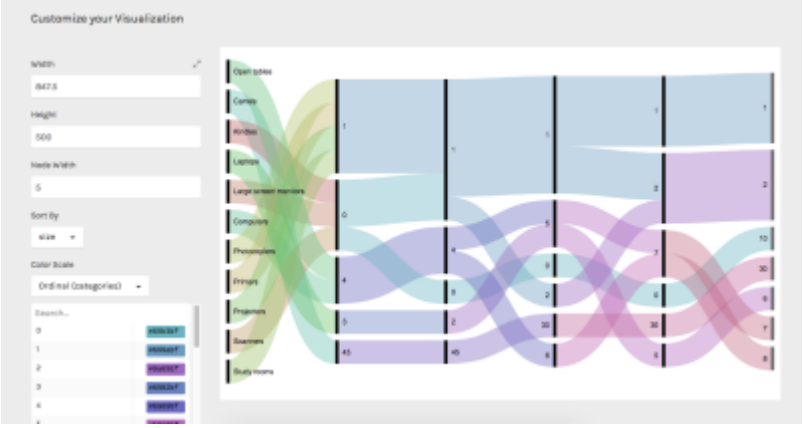


Después de este paso, vamos otra vez hacia abajo en la página y nos encontraremos con el generador de la gráfica según el diagrama de hayamos seleccionado.



Lo único que hace falta es arrastrar los bloques verdes de la izquierda, hacia las ventanas blancas de la derecha. Cabe notar, que las ventanas donde se arrastran los bloques nos indican qué tipo de dato debemos arrastrar ahi, si numeros o palabras. (numbers o strings).

Finalmente la app web nos enseña el gráfico generado de la data.

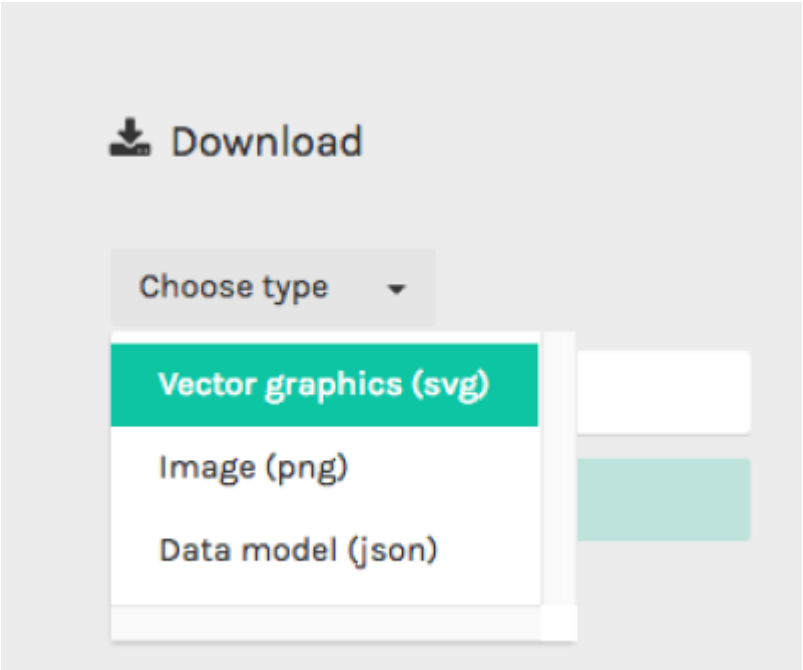


Pero esta grafica no nos sirve, porque suele ser un png o svg que se incrusta en una página web y que por lo general suele ser estática, sin ningún tipo de interactividad.

Para finalizar esta parte, hacemos scroll hacia la última parte de la página, donde se nos entregará la gráfica en formato svg.

A disposición en la parte izquierda, encontramos un panel donde podemos generar el json que necesitamos para generar el mismo gráfico pero usando d3 nativo.

Finalmente entonces, hacemos click en exportar json **“data model json”** y ya tendremos nuestra estructura de datos lista para inyectarle a la gráfica.



Generando la gráfica

Finalmente para generar la gráfica, insertamos el archivo json que nos da la web en la carpeta donde está el proyecto y lo importamos en el código javascript. Específicamente la línea 75, reemplazando el nombre por el nombre que tenga el json.

```
// load the data
d3.json("sankey.json", function(error, graph) {

  sankey
      .nodes(graph.nodes)
      .links(graph.links)
      .layout(32);
```