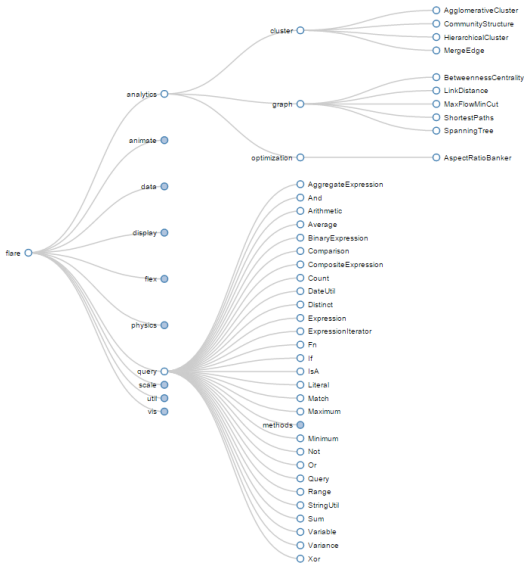
Tutorial D3.js: Collapsible Tree



Der sogenannte Collapsible Tree ist eine graphische Methode, um Hierarchien und Vernetzungen von Daten in einer Baumstruktur darzustellen. Die ausblendbare Baumstruktur wird mit einem Wurzelknoten initialisiert. Dieser enthält mehrere einblendbare Kinder-Knoten, die selbst wiederum als Elternknoten agieren können.

Datenaufbereitung

Das bestgeeignete Datenformat für diese Visualisierung ist das **JSON**(Javascript Object Notation)-Format, welches die hierarchische, vernetzte Form unterstützt. Ein solches Objekt besteht aus Attribut-Werte-Beziehungen, wobei die Werte eine Zeichenkette, eine Zahl, ein Objekt selbst oder eine Liste (Array) der aufgezählten Möglichkeiten sein können.

Beispiel:

```
{"Name": "Forschungsstelle Digitale Nachhaltigkeit", "Ort": "
    Bern", "Postleitzahl": 3012, "Leitung": {"Name": "Matthias
     Stuermer", "Titel": "Doktor"}, "Arbeitnehmer": [
  {"Name": "Janik Endtner", "Titel": "Student"},
  {"Name": "Oscar Meier", "Titel": "Student"}
  ]
}
```

Ein JSON-Objekt ist durch die geschweiften Klammern `{ }` gekennzeichnet. Eine Liste (Array) wird mit den eckigen Klammern `[ ]` definiert. Die Beziehungen werden durch den Doppelpunkt erstellt, wobei jeweils links davon das Attribut und rechts der entsprechende Wert ist (`{"Attribut": "Wert"}`). Da der Wert wie im Beispiel unter dem Attribut `"Leitung"` wieder ein Objekt selbst oder unter dem Attribut `"Arbeitnehmer"` auch eine Liste von Objekten sein kann, eignet es sich sehr gut für hierarchische Strukturen. Ein JSON-Objekt für eine Baumstrukturvisualisierung mit Eltern-, Kinder- und einem Wurzelknoten würde beispielsweise so aussehen:

```
{"Id": 1, "Name": "Wurzelknoten", "Kinder": [
  {"Id": 2, "Name": "Kind1", "Kinder": [
    {"Id": 3, "Name": "Kind1.1"}
  ]},
  {"Id": 4, "Name": "Kind2", "Kinder": [
    {"Id": 5, "Name": "Kind2.1", "Kinder": [
      {"Id": 6, "Name": "Kind2.1.1"},
      {"Id": 7, "Name": "Kind2.1.2"}
    ]},
    {"Id": 9, "Name": "Kind2.2"}
  ]}
]}
```

Da die Daten meistens im CSV-Format zur Verfügung stehen, müssen diese zuerst noch ins JSON-Format konvertiert werden. Das obige Beispiel könnte aus den folgenden einfachen CSV-Zeilen erstellt worden sein:

Id	Name	Elternknoten-Id
1	Wurzelknoten	-1
2	Kind1	1
3	Kind1.1	2
4	Kind2	1
5	Kind2.1	4
6	Kind2.1.1	5
7	Kind2.1.2	5
8	Kind2.2	4

Ein möglicher Konvertierungsalgorithmus würde zuerst ein Wurzelknoten-JSON-Objekt erstellen. Danach würde er einen Knoten nach dem anderen durchgehen und jede Zeile, deren Elternknoten-Id der Id des Wurzelknoten entspricht als Kind hinzufügen. Dieser Vorgang würde danach für alle gesetzten Kinder durchiteriert werden.

Code

Der Skriptcode kann unter dem Tab *Code* kopiert werden. Das Skript wird Schritt für Schritt erklärt, um sich einen Eindruck verschaffen zu können, wie die D3.js Library bezüglich des Collapsible Tree's funktioniert.

```
var margin = {top: 20, right: 120, bottom: 20, left: 120},
width = $(window).width() - margin.right - margin.left,
height = $(window).height() - margin.top - margin.bottom;

var i = 0,
duration = 750,
root;

var tree = d3.layout.tree()
.size([height, width]);

var diagonal = d3.svg.diagonal()
  .projection(function (d) {
  return [d.y, d.x];
  });

var svg = d3.select("#divCollapsibleTree").append("svg")
  .attr("width", width + margin.right + margin.left)
  .attr("height", height + margin.top + margin.bottom)
  .append("g")
  .attr("class", "drawarea")
  .append("g")
  .attr("transform", "translate(" + margin.left + "," +
     margin.top + ")");
```

Codeblock 1: Initialisierungen

Im ersten Teil werden die nötigen Initialisierungen durchgeführt. Der `margin` wird verwendet, damit die Visualisierung nicht die ganze Ansicht füllt und auf schöne Art und Weise zentriert werden kann. Die JQUERY-Bibliothek wird verwendet, um die Breite `$(window).width()` und die Höhe `$(window).height()` des Fensters zu bestimmen. Das `i` wird später als Iterationszähler verwendet. Auf die Variablen `duration` und `root` wird später noch genauer eingegangen. Sehr wichtig ist die grundlegende Initialisierung der Baumstruktur über den Befehl `d3.layout.tree().size([height, width])`. Hier übernimmt die D3.js Library im Hintergrund die Erstellung und wird später die Position der Knoten bezüglich der gesetzten Dimensionen setzen. Die Diagonale `diagonal` wird als Verbindungslinie zwischen den Knoten agieren. In einem SVG befindet sich der Koordinatennullpunkt oben links. Die x-Achse zeigt nach rechts, die y-Achse nach unten. Da das Tree-Layout normalerweise eine vertikale Aus-

richtung hat, werden hier im Skript die x- und y-Werte absichtlich vertauscht, um somit eine horizontale Ausrichtung zu erreichen. Eine genauere Beschreibung dazu gibt es unter dem Link `http://bl.ocks.org/mbostock/3184089`. Damit die Knoten und Linien überhaupt erst in das HTML integriert werden können, muss ein **SVG** (Scalable Vector Graphics) erstellt werden. Über den Befehl `append` wird das SVG dem HTML-Body mit den gewünschten Dimensionen hinzugefügt. Das SVG-Element `g` bezeichnet eine Gruppe, in welcher verschiedene Elemente zusammengefasst und wie hier zu erkennen auf die selbe Art und Weise transformiert werden können (`"translate("+ margin.left + ","+ margin.top + ")`). Diese Verschiebung wird verwendet, um das Ganze zu zentrieren und den Rand zu definieren. Das erste Gruppenelement mit der Klasse `"drawarea"` wird später für die Drag- und Zoom-Erweiterung verwendet.

Nach den ersten Initialisierungen werden die Daten eingelesen:

```
d3.json("/json/university-structure.json", function(error,
    flare) {
if (error) throw error;

root = flare;
root.x0 = height / 2;
root.y0 = 0;

function collapse(d) {
  if (d.children) {
  d._children = d.children;
  d._children.forEach(collapse);
  d.children = null;
  }
}

root.children.forEach(collapse);
update(root);
});
```

Codeblock 2: Einlesen der JSON-Daten

Über den Befehl `d3.json()` werden die JSON Daten eingelesen, und, falls es keinen Error gibt, in der Variable `flare` gespeichert. Das gesamte JSON-Objekt wird als Wurzelknoten (`root`) gesetzt. Zusätzlich werden die x- und y- Koordinaten für die Position des Knotens mit `height / 2` und `0` definiert. Der Wurzelknoten `root` befindet sich nach dieser Initialisierung genau vertikal zentriert ganz links in der Gruppe `g`.

Sehr interessant ist die Methode `collapse(d)`: Hier werden die zu Beginn ein-

geblendeten Knoten definiert. Wichtig ist zu erwähnen, dass das Baumstrukturlayout `tree`, das in Codeblock 1 definiert worden ist, in einer später vorgestellten Funktion die `children`-Attribute der Daten durchsucht, um alle eingeblendeten Knoten zu visualisieren. Deshalb wird in der Methode `collapse(d)` durch jedes `children`-Attribut durchiteriert, wobei dieses Attribut jeweils dem von dem `tree` unbeachteten `_children`-Attribut zugewiesen wird, um es in dieser Variable zwischenzuspeichern. Danach wird das `children`-Attribut `null` gesetzt, sodass der `tree` es nicht mehr für sein Layout berücksichtigt. Somit bewirkt der Aufruf `root.children.forEach(collapse)`, dass jeder Kinder-Knoten mitsamt all den weiteren Kinder-Knoten kein `children`-Attribut mehr hat und somit vom `tree` durch den später definierten Funktionsaufruf nicht mehr in die sichtbaren Knoten miteinbezogen wird.

Als Nächstes wird auf die wichtigste Methode `update(source)`, die, die den Baum erst visualisiert, eingegangen.

```
// Compute the new tree layout.
var nodes = tree.nodes(root).reverse(),
links = tree.links(nodes);

// Normalize for fixed-depth.
nodes.forEach(function(d) { d.y = d.depth * 180; });

// Update the nodes...
var node = svg.selectAll("g.node")
  .data(nodes, function(d) { return d.id || (d.id = ++i); });
```

Codeblock 3: Erstellen der Knoten und Linien

In diesem Abschnitt werden die eingeblendeten Knoten und Verbindungslinien über den vorher initialisierten `tree` ermittelt. Der Aufruf `tree.nodes(root).reverse()` durchsucht, wie oben beschrieben, alle `children`-Attribute und gibt die Knoten (`nodes`) für das Layout zurück. Technisch betrachtet besteht die `nodes`-Variable aus einer Liste, die alle Knotenobjekte enthält. Hier wird jedem Knoten automatisch noch ein `depth`-Attribut zugewiesen, was Auskunft über die hierarchischen Beziehungen gibt. So können die y-Werte (Beachte, dass die x- und y-Werte im ganzen Skript vertauscht worden sind) der Knoten abhängig von dieser Tiefe über den Aufruf `nodes.forEach(function(d){ d.y = d.depth * 180; })` gesetzt werden. Die Funktion `tree.links(nodes)` gibt die Verbindungslinien (`links`) zwischen den Knoten zurück.

Die letzte Zuweisung, die Selektion, ist eine der grundlegendsten Funktionalitäten von D3.js. Hier werden über die Methode `selectAll("g.node")` alle SVG-Elemente in der Gruppe `g` mit der CSS-Klasse `node` ausgewählt und mit

dem Datenarray `nodes` assoziert. Über die Methode `function(d){ return d.id || (d.id = ++i); }` werden zusätzlich noch Knoten-Id's gesetzt.

In D3.js gibt es drei **Selektionstypen**: Die **Enter**-, die **Update**- und die **Exit**-Selektion. Da im Moment noch keine Elemente der Klasse `node` existieren, können noch keine SVG-Elemente mit den Datenobjekten assoziert werden. Die Enter-Selektion kreiert für jedes noch nicht bestehende SVG-Element einen Stellvertreter, der mit dem entsprechenden Datenobjekt assoziert wird.

```
// Enter any new nodes at the parent's previous position.
var nodeEnter = node.enter().append("g")
  .attr("class", "node")
  .attr("transform", function(d) { return "translate(" +
     source.y0 + "," + source.x0 + ")"; })
  .on("click", click);

nodeEnter.append("circle")
  .attr("r", 1e-6)
  .style("fill", function(d) { return d._children ? "
     lightsteelblue" : "#fff"; });

nodeEnter.append("text")
  .attr("x", function(d) { return d.children || d._children ?
      -10 : 10; })
  .attr("dy", ".35em")
  .attr("text-anchor", function(d) { return d.children || d.
     _children ? "end" : "start"; })
  .text(function(d) { return d.name; })
  .style("fill-opacity", 1e-6);
```

Codeblock 4: Enter-Selektion

Der erste Aufruf `node.enter().append("g")...` bewirkt, dass für jeden Stellvertreter ein Gruppenelement `g` der Klasse `node` erstellt wird und auf die Position des `source`-Knoten, der beim ersten Durchgang der `update(source)`-Funktion der `root`-Knoten ist, verschoben wird. Falls auf diese Gruppe geklickt wird, wird die im späteren Verlauf definierte Funktion `click` aufgerufen. Im unteren Aufruf wird dann jeder neu hinzugefügter Gruppe ein `circle` hinzugefügt, das `"lightsteelblue"` eingefärbt wird, falls der assoziierte Knoten noch weitere Kinder enthält, und `"#fff"` sonst. Im letzten Teil werden jeweils noch die dazugehörigen Texte hinzugefügt. Man kann sich den Aufruf `node.enter()` als Iteration durch alle Stellvertreter vorstellen, wobei der Parameter `d` in den Funktionen sich jeweils auf das Datenobjekt des aktuellen Stellvertreters der Iteration bezieht. Allgemein bezieht sich das `d` in den Funktionsaufrufen bei allen Selektionen auf das momentane Element der Iteration.

Die Update-Selektion ist diejenige, die zu Beginn mit den Daten ausgewählt worden ist (`svg.selectAll("g.node").data(nodes, ...)`). Sei $n \in \mathbb{N}$ die Anzahl der bereits existierenden SVG-Elementen und $n <= m \in \mathbb{N}$ die Grösse des Datensets (`nodes`). Dann werden die ersten n Datenobjekte mit den n SVG-Elementen assoziert und in der Update-Selektion ausgewählt; Die $q = m - n$ noch nicht existierenden SVG-Elemente gehören zur Enter-Selektion.

```
// Transition nodes to their new position.
var nodeUpdate = node.transition()
  .duration(duration)
  .attr("transform", function(d) { return "translate(" + d.y
     + "," + d.x + ")"; });

nodeUpdate.select("circle")
  .attr("r", 4.5)
  .style("fill", function(d) { return d._children ? "
     lightsteelblue" : "#fff"; });

nodeUpdate.select("text")
  .style("fill-opacity", 1);
```

Codeblock 5: Update-Selektion

Bei der Update-Selektion kann über den Befehl `transition()` zusätzlich eine Animation mit einer bestimmten Dauer `duration(duration)` erzeugt werden. Die Anpassungen der bereits bestehenden SVG-Elementen geschieht in diesem Fall dynamisch und animiert. Da die Enter-Selektion vor der Update-Selektion durchgeführt worden ist, besitzt zum jetzigen Zeitpunkt jedes Datenobjekt ein assoziertes SVG-Element. Deshalb bewegt sich jedes Element von der Position des `source`-Knoten - Die Position der Kinder-Knoten ist in der Enter-Selektion im Codeblock 4 auf die des `source`-Knoten gesetzt worden - hin zu seiner eigentlicher Position `function(d){ return "translate( "+ d.y + ","+ d.x + ")"; }`. Die `circle` wachsen während der Animation von der gegebenen Grösse (Codeblock 4) $1e-6$ auf einen Radius `"r"` von 4.5. Der Text wird über das Style-Attribut `"fill-opacity"` sichtbar gemacht.

Die letzte Selektion, die Exit-Selektion, beinhaltet alle SVG-Elemente, für welche kein entsprechendes Datenobjekt mehr existiert. Diese werden normalerweise entfernt.

```
// Transition exiting nodes to the parent's new position.
var nodeExit = node.exit().transition()
  .duration(duration)
  .attr("transform", function(d) { return "translate(" +
     source.y + "," + source.x + ")"; })
  .remove();
```

```
nodeExit.select("circle")
  .attr("r", 1e-6);

nodeExit.select("text")
  .style("fill-opacity", 1e-6);
```

Codeblock 6: Exit-Selection

Hier werden die zu entfernenden, nicht mehr im Datenset existierenden Knoten, zuerst über eine Animation auf die Position des `source`-Knoten transformiert, bevor sie über den Befehl `remove()` vom dem SVG entfernt werden. Bevor diese Elemente verschwinden, werden jeweils während der Animation über die unteren beiden Funktionen noch die Kreisradien verkleinert und der Text ausgeblendet.

Jetzt muss dieselbe Prozedur noch für die Verbindungslinien durchgeführt werden.

```
// Update the links...
var link = svg.selectAll("path.link")
  .data(links, function(d) { return d.target.id; });

// Enter any new links at the parent's previous position.
link.enter().insert("path", "g")
  .attr("class", "link")
  .attr("d", function(d) {
    var o = {x: source.x0, y: source.y0};
    return diagonal({source: o, target: o});
});

// Transition links to their new position.
link.transition()
  .duration(duration)
  .attr("d", diagonal);

// Transition exiting nodes to the parent's new position.
link.exit().transition()
  .duration(duration)
  .attr("d", function(d) {
var o = {x: source.x, y: source.y};
  return diagonal({source: o, target: o});
  }).remove();
```

Codeblock 7: Bearbeiten der Verbindungslinien

In der Enter-Selektion werden hier Pfade generiert, die durch die im Codeblock 1 Funktion `diagonal` bestimmt werden. Diese Funktion erwartet als Input ein JSON-Objekt, welches das `source`- und das `target`-Objekt beinhal-

tet. Diese Objekte bestehen lediglich aus x- und y-Koordinaten. Der Aufruf `return diagonal({source: o, target: o})` kreiert automatisch eine Verbindungslinie zwischen `source` und `target`. Ähnlich wie bei den Knoten, werden die Verbindungslinien so initialisiert, dass `source = target` ist. Das bedeutet, das diese reflexiv zum eigenen Knoten führen. In der Update-Selektion werden die `links`, was technisch gesehen eine Liste von `{source: firstNode, taget: secondNode}` ist, durch die `diagonal`-Funktion abgebildet aufgrund der `transition()` animiert. Die Exit-Selektion verläuft wiederum ähnlich wie bei den Knoten. Zuerst werden die Verbindungslinien reflexiv auf den `source`-Knoten gesetzt und danach entfernt.

Im letzten Abschnitt der `update(source)`-Funktion werden die momentanen Positionen der Knoten in den Koordinaten `x0` und `y0` zwischengespeichert, so dass sie für die Erstellung, wie in den obigen Enter-Selektionen, verwendet werden können, sodass die Animation der darauffolgenden Update-Selektion an der richtigen Position startet.

```
// Stash the old positions for transition.
nodes.forEach(function(d) {
  d.x0 = d.x;
  d.y0 = d.y;
});
```

Codeblock 8: Zwischenspeichern der momentanen Positionen

Ein wichtiger Bestandteil der Visualisierung, die Interaktion, fehlt noch. Der in Codeblock 4 definierte Listener `on("click", click);` erwartet noch eine Implementierung der `click`-Funktion.

```
// Toggle children on click.
function click(d) {
if (d.children) {
  d._children = d.children;
  d.children = null;
} else {
  d.children = d._children;
  d._children = null;
}
update(d);
}
```

Codeblock 9: Implementierung der Interaktion

Da der `tree` die Knoten über die Funktion `tree.nodes(root).reverse()` und die Verbindungslinien über `tree.links(nodes)` (siehe Codeblock 3) automatisch durch Iterieren über alle `children`-Attribute kreiert, kann die `click`-Methode

auf einfache Art und Weise implementiert werden: Falls der angeklickte Knoten ein `children`-Attribut hat (Kind-Knoten wird momentan eingeblendet), wird dieses im `_children`-Attribut zwischengespeichert und danach auf `null` gesetzt. Das Umgekehrte wird durchgeführt, falls das `children`-Attribut nicht existiert. Nach der Anpassung der Attribute wird `update(d)` aufgerufen, wobei der angeklickte Knoten als `source` agiert und die Animationen von diesem aus starten (Update) resp. bei diesem enden (Exit).

Eine detailliertere Beschreibung zu den D3-Selektionen ist unter `https://medium.com/@c_behrens/enter-update-exit-6cafc6014c36#.8v659zot0` zu finden.

Die Drag- und Zoom-Erweiterung ist dank integrierter D3.js Funktionalität schnell umgesetzt. Der folgende Code reicht aus, um die Erweiterung zu implementieren.

```
//Add zoom extension
d3.select("svg")
  .call(d3.behavior.zoom()
    .scaleExtent([0.5, 5])
    .on("zoom", zoom));

function zoom() {
  var scale = d3.event.scale,
  translation = d3.event.translate,
  drawAreaWidth  = d3.select(".drawarea").node().getBBox().
     width,
  tBound = -height * scale + 50 * scale,
  bBound = height - 50 * scale,
  lBound = (-drawAreaWidth) * scale + 50 * scale,
  rBound = width - 50 * scale,
  // limit translation to thresholds
  translation = [
    Math.max(Math.min(translation[0], rBound), lBound),
    Math.max(Math.min(translation[1], bBound), tBound)
  ];

  d3.select(".drawarea")
    .attr("transform", "translate(" + translation + ")" +
    " scale(" + scale + ")");
}
```

Codeblock 10: Drag- und Zoom-Erweiterung

Über die `call`-Funktion wird der Zoom-Listener initialisiert. Sobald ein Zoom oder ein Drag erfolgt, wird die `zoom`-Funktion aufgerufen. Über `d3.event.scale` resp. `d3.event.translation` werden der momentane Zoom und die Verschie-

bung ermittelt. Die weiteren Initialisierungen werden verwendet um Letzteres auf den sichtbaren Bereich einzuschränken, so dass der Nutzer die Baumstruktur nicht vollständig aus dem Container schieben kann. Durch den letzten Aufruf wird das `"transform"`-Attribut der Gruppe `"drawarea"` (in Codeblock 1 initialisiert) verändert, was den visuellen Drag- und Zoom-Effekt generiert.