UROP: Network Visualisation

At the UK Dementia Research Institute (UKDRI), the investigation into neuroscience generally involves hundreds and thousands of data sets collected from a living lab over a period of several years. Everyday, researchers scan through arrays of numbers in these data sets in order to identify and highlight the significance of particular abnormalities and trends. However, the tasks can be highly repetitive and time-consuming due to the sheer size of the data sets. Hence, as the primary assignment of this UROP placement, a network visualisation tool native to Jupyter Notebook is created under the supervision of Dr.Eyal Soreq. This report will highlight on the creation of the network visualisation library.

Network Visualisation

A network consists of distinct nodes that each contains unique information, connected to each other by links or edges. For instance, nodes can represent the people in a class, countries or the activities carried out by an individual in a day. The relationships between nodes is attributed to the links, which may include the strength of connection, source and target of link, and type of connection.

Apart from being a data format, a network is generally used to identify patterns that are displayed in the entire or part of the network. To illustrate, the positions of the nodes in a visual network may reveal the underlying relationships between these nodes as opposed to the other nodes.

In this context, the nodes may represent the daily activity carried out by an individual throughout the day, with the links representing the probability of transitioning to one node to another. The probabilities are calculated based on years of data collected from a living lab and arranged in the form of a transition matrix. It ultimately demonstrates the change in the behaviour of an individual over the years, which can potentially exhibit early indicators of dementia.

The network visualisation tool will be used to visualise this Markovian System where the probability of transitioning from one state to the next is solely dependent on the previous state (Markovian chain without memory), given in the form of a transition matrix. To understand the format of a transition matrix, one is given in the form of a table below:

States	From A	From B
To A	0.8	0.3
To B	0.2	0.7

As shown in the table, the columns represent the source states whereas the rows represent the target states. For instance, the probability of moving from state A to state B is 0.2, and moving from state A to state A again is 0.8.

Visualisation Model Criteria

In order to support the users, the visualisation model must be equipped with the following features:

1. Nodes to represent state space.
2. Bidirectional arrows (including self loops) to indicate network connections and relationships between states.
3. Node position mapping based on links and relationships with other nodes.
4. Interactivity with user (user controlled hovering events).

Selection of Library

As the native programming language of Jupyter Notebook is Python, the two common Python graph plotting libraries, Matplotlib and Bokeh, have come into consideration. However, the implementation using Matplotlib lacks the interactivity with user controlled events while the Bokeh framework has limited flexibility in demonstrating the bidirectional property of the network despite incorporating NetworkX library.

Under the recommendation of my supervisor, the D3 library is explored, with reference to **this website** which features an example of visualising the Markov Chain with D3. D3 supports a highly customisable array of visual elements along with a wide variety of interactive events with the user.

Injecting Javascript into Jupyter Notebook

As D3 is a Javascript library and Jupyter Notebook runs on Python by nature, the Jupyter Notebook cell magic **%%javascript** is used to write Javascript in the notebook. In order to access D3, the notebook comes with a baked-in RequireJS which allows us to import the D3 module.

An example of implementing D3 in Jupyter notebook is given below.

```
%%javascript

require.config({
    paths: {
        d3: 'https://d3js.org/d3.v6.min'
    }
});

(function(element) {
    require(["d3"], function(d3) {
        //insert D3 script here;
    })
})(element)
```

Define Data Sets

To create simulated data sets, a sample transition matrix has to be generated using the code below. It incorporates the element of noise to mimic real data sets. The key feature of a transition matrix is that each column's sum must add up to 1.

```
import numpy as np

def sample(states_num: int = 4,
           transition_prob: float = None,
           noise_scale: float = 0.01,
           seed: int = 2021):

    rng = np.random.default_rng(seed)
    if type(transition_prob)==type(None):
        transition_prob = softmax(rng.normal(size=(states_num,states_num)))
    noise = rng.normal(0,noise_scale,(states_num,states_num))
    sample_prob = transition_prob + noise
    negative_noise = sample_prob.min(axis=0)
    sample_prob = sample_prob + (negative_noise<0)*np.sign(negative_noise)*negative_noise
    sample_prob = sample_prob/sample_prob.sum(axis=0)
    return sample_prob

def softmax(x):
    """Compute softmax values for each sets of scores in x."""
    return np.exp(x) / np.sum(np.exp(x), axis=0)
```

Using the transition matrix, a list of nodes and a list of links with attributes are generated as below, which form the basis of the data sets. If a list of state names or images is provided, it can be added onto the nodes as an attribute.

```
import string

def nodesCalibration(transitionMatrix, state_name, image):
    lengthMatrix = len(transitionMatrix[0])

    # if state_name is not given, returns a list of alphabets
    if type(state_name) == type(None):
        state_name = string.ascii_uppercase[:lengthMatrix]

    # if image is not given, returns a list of Nones
    if  type(image) == type(None):
        image = [None]*lengthMatrix

    nodes = {}
    for i in range(lengthDict):
        nodes.append({"id":state_name[i],"index":i,"image":image[i]})
    return nodes

def linksCalibration(transitionMatrix):
    lengthMatrix = len(transitionMatrix[0])
    links = []

    for j in range(lengthMatrix):
        for i in range(lengthMatrix):
            if transitionMatrix[k][j][i] != 0:
                links.append({"source": i,"target": j,"id": str(i)+str(j),"weight":
    transitionMatrix[j][i]})
            else:
                links.append({"source": i,"target": j,"id": str(i)+str(j),"weight": -1})
    return links
```

Nodes and Links Positioning

To compute the locations of the nodes, a static D3 force simulation layout is utilised as it defines the location of the nodes based on several force parameters such as link forces, charge forces and gravitational forces.

To set up the force simulation, the data sets have to be passed into Javascript using JSON. An example of it would be to save the data set in a separate JSON text file and passing it into the Javascript function as shown.

```
import json

#example of nodes data set
nodes = [{'id': 'A', 'index': 0,'image': None},
        {'id': 'B', 'index': 1,'image': None}]

# open and write into the file nodes.json
with open('nodes.json', 'w') as outfile:
    json.dump(nodes, outfile)
```

```
%%javascript
require.config({
    paths: {
        d3: 'https://d3js.org/d3.v6.min'
    }
});

// element is the placeholder as it refers to the output of the notebook
$.getJSON("nodes.json", function(jsonnodes) {
$.getJSON("links.json", function(jsonlinks) {
(function(element) {
        require(['d3'], function(d3) {
            const nodes = jsonnodes, links = jsonlinks,
                  width = 500, height = 500;

            const simulation = d3.forceSimulation(nodes,d=>d.id)
                 .force("link",d3.forceLink().links(links,d=>d.id)
                 .distance(d => 180-d.weight*150))
                .force("charge", d3.forceManyBody().strength(-200))
                .force('collision', d3.forceCollide().radius(4 * 10))
                .force("center", d3.forceCenter(width / 2, height / 2))
                .stop();

            simulation.tick(200); // to allow the nodes to render for 200 ticks
        })
})(element)
})})
```

After the simulation is called, the nodes and links will have the attributes 'x', 'y', 'vx', 'vy' appended to them. They represent the positions and velocities of the elements at 200 ticks as defined in the code. These attributes will provide the coordinates of the visual 'nodes' and 'links' that will be created later.

In addition, the simulation call also updates the sources and targets of the links with the updated nodes.

Creating Node Elements

Using the D3 enter() command, individual visual elements can be created easily with the data set available. Firstly, create a Scalable Vector Graphic (SVG) element where all the visual elements will be added to.

Then, select all the 'g' elements and take the nodes as the data set. If there are existing 'g' elements, the data will be bound to them. However, in the event that there is more data than there is 'g' elements, the **.enter()** and **.append('g')** command will append new 'g' element bound to the remaining data.

In the previous blocks, we defined nodes to contain the information of two nodes. Hence, two circle elements that are bound to the two data elements are created with the **.append('circle')** command. The attributes of the circle element can then be altered as shown.

```
const svg = d3.select(element.get(0)) // selecting the output of notebook
            .append("svg")
            .attr("width", width)
            .attr("height", height);

const node = svg.append("g")
            .attr("class", "nodes")
            .selectAll("g")
            .data(nodes)
            .enter()
            .append("g");

// circular nodes
const circle = node.append("circle")
            .attr("r", 10)           //radius of circle
            .attr("fill","blue")     //fill of circle
            .attr("cx", d => d.x)    //x coordinate of centre
            .attr("cy", d => d.y)    //y coordinate of centre
```

An example of the notebook output is shown below.

Figure 1: Node Output Example

Creating Link Elements

By default, the link between nodes is a straight line which does not provide informative features regarding the attributes of the link. On top of that, all the links between two nodes are overlapped, and there is no way to distinguish self loops (node links to itself). An example of this can be observed on **this example** by Mike Bostock.

A good example of the intended features can be seen on **this example** which showcases a network of mobile patent lawsuits among companies, again by Mike Bostock. It contains bidirectional arrows that clearly indicate the relationships between nodes, and their colours are based on the types of the lawsuits. However, the main feature is the curved links which allow us to observe the two way relationships without them overlapping on each other.

Arrows

Based on the inspiration from the example, modifications are made to adapt the visualisation to our intended features. With the aid of markers, arrowheads can be added to the end of the links to indicate direction of the link. In the code below, two types of arrowheads are defined as it improves the clarity for self loop arrows. The colour constant binds each node id to one of the 10 colours in the d3.schemeCategory10. This will be used to identify links coming from the same node.

```
const types = Array.from(new Set(nodes.map(d => d.id))),
      colour = d3.scaleOrdinal(types,d3.schemeCategory10);

const arrows = svg.attr("class", "arrow")
                        .selectAll('.arrows') //set arrow marker
                        .data(types)
                        .enter()
                        .append("g")

//for loops away from self
const arrow_away = arrows.append("defs")
                       .append("marker")
                        .attr("id", d => `arrow-${d}`)
                        .attr("viewBox", "0 -10 20 20")
                        .attr("refX", 1)
                        .attr("refY", 0)
                        .attr("markerWidth", 3)
                        .attr("markerHeight", 3)
                        .attr("orient", "auto-start-reverse")
                        .append("path")
                        .attr("fill", colour)
                        .attr("d", "M0,-10L20,0L0,10");

//for self loops
const arrow_self = arrows.append("defs")
                       .append("marker")
                        .attr("id", d =>`arrow2-${d}`)
                        .attr("viewBox", "0 -10 20 20")
                        .attr("refX", 10)
                        .attr("refY", 2)
                        .attr("markerWidth", 3)
                        .attr("markerHeight", 3)
                        .attr("orient", "auto-start-reverse")
                      .append("path")
                        .attr("fill", colour)
                        .attr("d", "M0,-10L20,0L0,10");
```

Link Paths

To draw curved links between nodes, the path of the link is computed by the function **curvepath()**, whereas the function **shortenedpath()** returns a shortened curve path in order to improve clarity of the network as the number of links connected to each nodes increases.

The **shortenedpath()** function also identifies and defines the paths of self loops as they differ from the other links.

```
//function to define curved link path
function curvepath(d){
    var dx = d.target.x - d.source.x,
        dy = d.target.y - d.source.y,
        dr = Math.sqrt(dx **2 + dy **2);
    return "M" + d.source.x + "," + d.source.y + "A" + dr + ","
               + dr + " 0 0,1 " + d.target.x + "," + d.target.y;
};

//function to define link path 2.0
function shortenedpath(d){
    if (d.source == d.target) {
       var xRotation = -45,
           largeArc = 1,
           sweep = 1,
           drx = 15,
           dry = 15;
       var dx1 = d.source.x+10,
           dy1 = d.source.y+10,
           dx2 = d.target.x+10,
           dy2 = d.target.y-10;

        return "M" + dx1 + "," + dy1 + "A" + drx + "," + dry + " " +
               xRotation + ","+ largeArc + "," + 0 + " " + dx2 + "," + dy2;
    } else {
        // length of current path
        var pl = this.getTotalLength(),
        // adjust r for suitable radius
            r = 25.6568,
        // position close to where path intercepts circle
            m = this.getPointAtLength(pl - r), //approaching end
            m2= this.getPointAtLength(r+2); //leaving end

        var dx = m.x - d.source.x,
           dy = m.y - d.source.y,
           dr = Math.sqrt(dx * dx + dy * dy);

        return "M" + m2.x + "," + m2.y + "A" + dr + "," +
              dr + " 0 0,1 " + m.x + "," + m.y;
}};
```

Create Links

Combining the arrows and the link paths, visual links can be created using the same method as the nodes. To visually identify the strength of the connections, the widths of the links are defined by the probability of the transition.

The **markerType()** function identifies if the link is a self loop when adding the arrowhead marker to the link.

```
var link = svg.append("g")
            .attr("class", "links")
            .selectAll("path")
            .data(links,d=>d.id)
            .enter().append("g");

var path = link.append("path")
            .attr("fill","none")
            .attr("stroke-width",function(d){
                //set width to 0 if link has 0 probability
                if (d.weight <= 0){return 0}
                //scale the width for better contrast
                else{return 2+4*d.weight}})
            .attr("opacity",0.3)
            .attr("marker-end", markerType)
            .attr("d",curvepath)
            .attr("d",shortenedpath)
            .attr("stroke",d=>colour(d.source.id));

function markerType(d){ //set different marker for self loop
            if (d.source == d.target) {
            return `url(${new URL(`#arrow2-${d.source.id}`,location)})`
            } else {
            return `url(${new URL(`#arrow-${d.source.id}`,location)})`
            }};
```

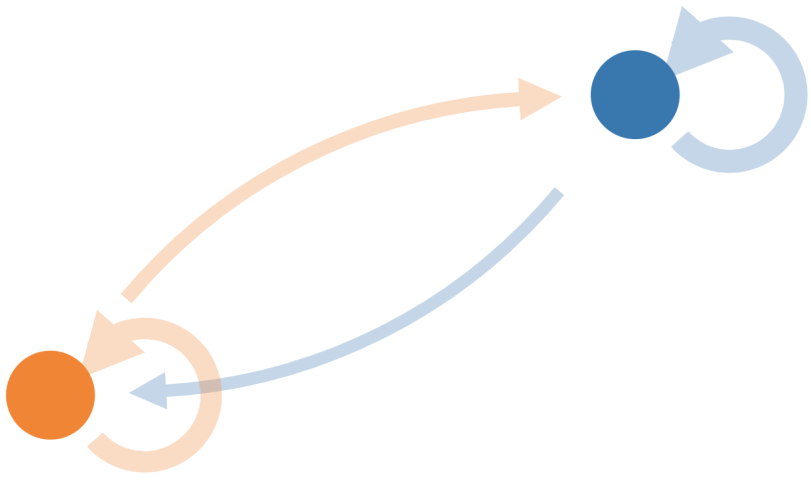

Figure 2: Nodes with Links Output Example

Interactivity

Using D3 event listeners, user controlled events such as button clicks, mouse hovering and sliders input can trigger certain actions within the visualisation. For our purposes, the mouse hovering event is used to inspect the attributes of the nodes or the links. On top of that, the links are set to enlarge to emphasise on the selected link.

To set the event listeners, the **.on('mouseover',action)** command is added after the links and nodes as shown below, where it calls a function when the cursor is hovering over it.

```
var circle = node.append("circle")
        .attr("r", 10)
        .attr('fill',d => colour(d.id))
        .attr("cx", d => d.x)
        .attr("cy", d => d.y)
        .on('mouseover', motionInNode)
        .on('mouseout', motionOutNode);

// create a 'g' element for hovering elements
const group = svg.append("g");

function motionInNode(d){
    var name = "", length = 0, locx = 0, locy = 0;

    d3.select(this).transition()
        .attr("x",d=>d.x-15)
        .attr("y",d=>d.y-15)
        .attr("width",30)
        .attr("height",function(d){
                name = d.id, locx = d.x, locy = d.y
                length = 10+10*d.id.length
                return 30});

    group.append("rect")
        .attr("rx", 6)
        .attr("ry", 6)
        .attr('height',20)
        .attr('width',length)
        .attr('id','stateRect')
        .attr('fill','black')
        .attr('opacity',0.5)
        .attr('x',locx-length/2)
        .attr('y',locy+22)

    group.append("text")
        .attr("id", 'stateshow')
        .attr('font-size',15)
        .attr('fill','white')
        .attr('x',locx)
        .attr('y',locy+37.5)
        .attr('text-anchor','middle')
        .text(name)
       };

function motionOutNode(d) {
    d3.select(this).transition()
                    .attr("x",d=>d.x-10)
                    .attr("y",d=>d.y-10)
                    .attr('height',20)
                    .attr('width',20);
    d3.select('#stateshow').remove();
    d3.select('#stateRect').remove();
};
```

As shown below, whenever the mouse hovers over the links, a text bubble will pop up, indicating the probability of the selected link while the link itself enlarges.

Whenever the nodes are selected, a text bubble indicating the state name pops up right underneath it. Text has also been added onto the nodes to indicate the different states.

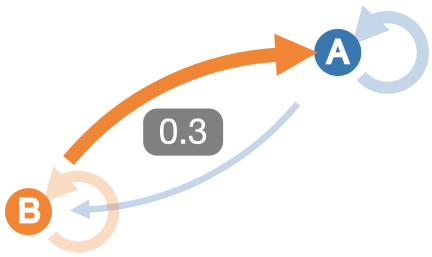

Figure 3: "Mouseover" Event Example

Insert Image on Nodes

To improve visual illustration, images can be added onto the nodes for better understanding. When we are generating the list of nodes, it can be seen that the nodes contain an attribute called image, which is a string of the URL to the image. This image attribute will provide the information needed when the element is created as below.

```
// add images to nodes if given
const image = node.append("svg:image")
        .attr('class','images')
        .attr("xlink:href",d => d.image)
        .attr("id", d =>`image-${d.id}`)
        .attr("x", d => d.x-10)
        .attr("y", d => d.y-10)
        .attr("height", 20)
        .attr("width", 20)
        .on('mouseover', motionInNode)
        .on('mouseout', motionOutNode);
```

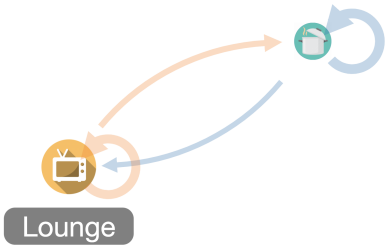

Figure 4: Node With Image Example

Styling with CSS

The elements created with D3 are commonly styled using CSS. It is a concise method to customise the visual parameters of the elements. An example of the CSS format is shown below. As CSS is written in the HTML language, the Jupyter notebook cell magic **%%html** can be called to write HTML codes in a notebook cell.

As CSS wraps over the D3 commands, only fixed variables are specified in CSS.

```
%%html
<style>
    .links path {
        fill: none;
    }
    .nodes circle {
        pointer-events: all;
        stroke: #fff;
        stroke-width: 2px;
        opacity: 1.0;
    }
    .nodes text {
        pointer-events: none;
        font: 10px sans-serif;
        fill: #fff;
        stroke-width: 0.6px;
        opacity: 1.0;
    }
</style>
```

Time Slider

To browse through data along a timeline, a time slider is created to manipulate the data selection. Various options have been explored with regards to the slider such as the Ipywidget library as well as the Panel library. As D3 is more compatible with HTML, an input slider is created in HTML which links directly to the D3 scripts that triggers a change whenever there is an input. The 'id' labels allow D3 to select the element just by its 'id'.

```
%%html
<p>Data Select:
<input type="range" min="0", max="%%matrixLength%%", step="1", value="0" id="dataSelect"/>
<output id="dataOutput">0</output>
</p>
```

It is important to note that since a series of data is now given (an array of matrices), the **nodesCalibration()** and **linksCalibration()** functions now compute the nodes and links in the form of a dictionary with the indices as the keys.

Whenever there is an input, the nodes and links data are selected from the dictionary depending on the value of the input. If the selected data has not been selected previously, the script will run the force simulation to compute their positions accordingly. Otherwise, it will call the data directly and carry on with the remaining codes.

Several update functions have been laid out to update the model whenever a new data set is selected.

```
// update chart when slider is moved
d3.select("input[type=range]#dataSelect").on("input", function() {
    var data;
        data = this.value;
        nodes = nodesData[data];
        links = linksData[data];

    // compute positions if fresh data is selected
    if (nodesData[data][0].x == undefined){
        const newSimulation = d3.forceSimulation(nodes,d=>d.id)
            .force("link", d3.forceLink().links(links,d=>d.id)
            .distance(d => link_distance-d.weight*150)                .strength(function(d){
                if (d.weight<=0){return 0} //strength adjusted to 0 for 0 probabilities
                else{return 1}}))
            .force("charge", d3.forceManyBody().strength(link_charge))
            .force('collision', d3.forceCollide().radius(collision_scale * node_radius))
            .force("center", d3.forceCenter(width / 2, height / 2))
            .stop();

        // allow simulation to run
        newSimulation.tick(%%ticks%%);
    };

    // update position of circles
    var newCircle = node.selectAll('circle')
       .data(nodes, d=>d.id)
       .transition()
       .duration(800)
       .attr("cx", d => d.x)
       .attr("cy", d => d.y)

    // update position of texts
    var newText = node.selectAll('text')
        .data(nodes, d => d.id)
        .transition()
        .duration(800)
        .attr("x", d => d.x)
        .attr("y", d => d.y)

    // update position of images
    var newImage = node.selectAll(".images") //because unable to select 'svg:image'
        .data(nodes, d => d.id)
        .transition()
        .duration(800)
        .attr("x", d => d.x-10)
        .attr("y", d => d.y-10)

    // update position of links
    var updateLink = d3.selectAll('.connections') //selectAll class doesnt work
        .data(links,d=>d.id)

    updateLink.attr('d',curvepath)
        .attr('d',shortenedpath)
        .attr('stroke-width',function(d){
            if (d.weight <= 0){return 0}
            else{return 2+link_width_scale*d.weight}});

    // update slider text
    d3.select("output#dataOutput").text("data set selected : " + data);
});
```

Zoom

Zoom is included in the visualisation to enable the users to inspect the elements in a magnified manner. However, some of the visual elements can be positioned outside of the output cell on initialisation as their positions are unrestrained. Hence, the **lapsedZoomFit()** function is created to initialise the zoom scale to fit all elements in the output cell.

```
const zoom = d3.zoom()
            .extent([[0, 0], [width, height]])
            .scaleExtent([0.2, 10])
            .on("zoom", zoomed);

//set zoom for model
function zoomed({transform}) {
    node.attr("transform",transform);
        link.attr("transform",transform);
        group.attr("transform",transform);
};

svg.call(zoom);

function lapsedZoomFit() {
    var bounds = svg.node().getBBox(),
        midX = bounds.x+bounds.width/2,
        midY = bounds.y+bounds.height/2;
        var scale = (0.75) / Math.max(bounds.width / width, bounds.height / height);
        var translateX = width / 2 - scale * midX;
        var translateY = height / 2 - scale * midY;
        svg.transition().duration(0).call(
            zoom.transform, d3.zoomIdentity
                .translate(translateX, translateY)
                .scale(scale)
        );
}

lapsedZoomFit();
```

Save SVG to PNG

To save the SVG generated into a local PNG file, the SVG element has to be converted to a string with several attributes altered. This is because all of the other elements are appended to the SVG tag, hence, they are all captured in the SVG string. Then, a canvas element is created with an image containing the information of the SVG element. Do note that a button has been created with the id **'saveButton'** to trigger the function whenever it is pressed.

```
d3.select("#saveButton").on("click", function(){
    var html = d3.select("#SVG")
        .attr("version", 1.1)
        .attr("xmlns", "http://www.w3.org/2000/svg")
        .node().parentNode.innerHTML;

    // console.log(html);
    var imgsrc = ''+ btoa(html);
    var img = '<img src="'+imgsrc+'">';
        d3.select("#svgdataurl").html(img);

    var canvas = document.createElement("canvas"),
        context = canvas.getContext("2d");
```

```
14
15      // default setting around 1.8, set to 7.2 for higher resolution
16      // const pixelRatio = window.devicePixelRatio || 1;
17      const pixelRatio = 7.2;
18
19      canvas.width = width * pixelRatio;
20      canvas.height = height * pixelRatio;
21      canvas.style.width = `${canvas.width}px`;
22      canvas.style.height = `${canvas.height}px`;
23      context.setTransform(pixelRatio, 0, 0, pixelRatio, 0, 0);
24
25      var image = new Image;
26          image.src = imgsrc;
27          image.onload = function() {
28              context.drawImage(image, 0, 0);
29
30              var canvasdata = canvas.toDataURL("image/png",1.0);
31
32              var pngimg = '<img src="'+canvasdata+'">';
33              d3.select("#pngdataurl").html(pngimg);
34
35              var a = document.createElement("a");
36              a.download = 'diagram_'+date_list[sliderDataValue]+'.png';
37              a.href = canvasdata;
38              a.click();
39          };
40  });
```

Link Pruning Slider

Inevitably, the data set collected contains noises that slightly distorts the data. Hence, to set up a threshold for links to appear visually, a link pruning slider is created to select the threshold. Any link with a weight equal to or below the threshold will be removed from the SVG, thus eradicating unwanted noise in the data.

```
1   d3.select("input[type=range]#linkThreshold").on("input", function() {
2       var threshold = this.value;
3
4       // select all links
5       var updateLink = d3.selectAll('.connections') //selectAll class doesnt work
6                          .data(links,d=>d.id)
7
8       // change link width to 0 if weight is below threshold
9       updateLink.transition()
10          .duration(800)
11          .attr('stroke-width',function(d){
12              if (d.weight <= threshold){return 0}
13              else{return 2+link_width_scale*d.weight}})
14
15      //  update slider text
16      d3.select("#linkThresholdOutput").text("Link Threshold : " + threshold);
17  });
```

Create Library

After the visualisation model is finalised, the Javascript and HTML files have to be wrapped in a callable function in Jupyter notebook. Hence,a Python module with the function **plot_force_directed_graph()** is written to call the files and present the output in a notebook cell. This function replaces keywords in the Javascript and HTML files with configured values to ease configuration. It is created with inspiration from **this repository**.

Several alterations have been added to the model for a better visual representation:

1. Node locations are only defined via force simulation if they are not given.
2. Nodes are now rounded rectangular text boxes with state names for clarity.
3. Hovering on the nodes magnifies the links that source from it while displaying a data box of the strengths of the links.
4. The 'hover-in' and 'hover-out' events for the nodes and links are disabled for 800 ms (equivalent to time taken to complete animation) whenever a new data set is selected through the slider to prevent system bugs.
5. CSS parameters are defined directly in D3 as the save function does not capture the features defined by it.

In addition, a 'datasort.py' module has been added to reformat data extracted from a CVS file into the correct model input format by utilising Pandas. The library can be found **here** with a sample notebook attached for demonstration purposes.

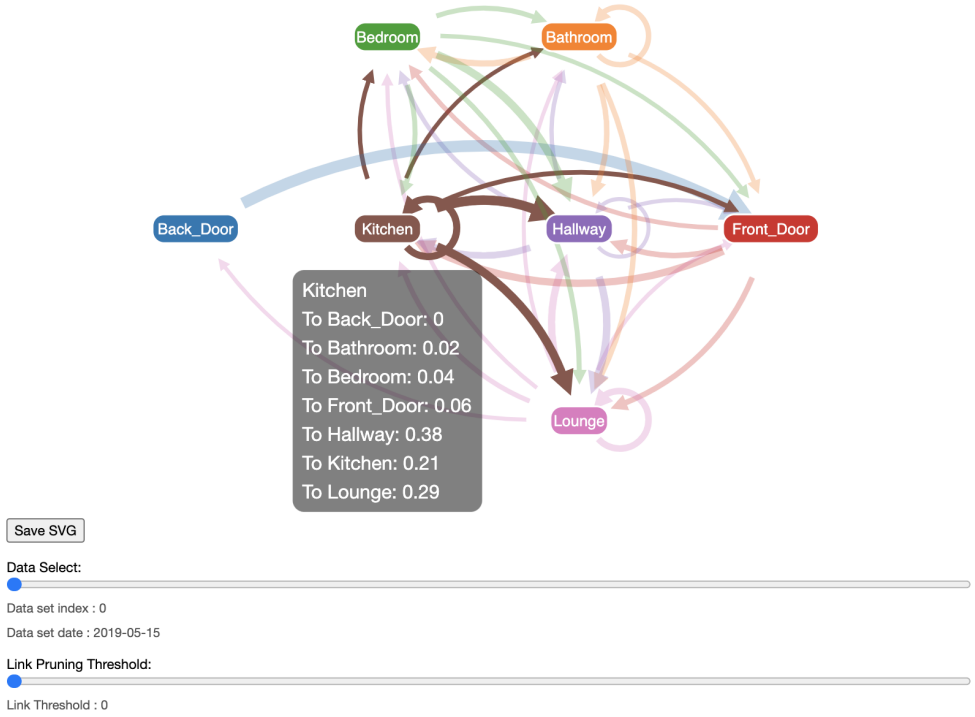

Figure 5: Sample Notebook Output Example

Conclusion

It is apparent that D3 is a very powerful library that allows its user to create highly customised graphs. On the same note, the seamless integration of Javascript and HTML into the Jupyter Notebook is undoubtedly the key to achieve this level of success for the model.

Inspecting the model with real data sets, it is certain that it provides a clear and direct way of inspecting the data without diving into the CSV files. Besides that, the time slider allows quick change of data sets.

In theory, the same library should work in Jupyter Lab. However, Jupyter Lab does not have a baked-in RequireJS. Thus, minor adjustment has been made to load D3 into the script without using RequireJS, and it is reflected in the repository under the Jupyter Lab folder.

Acknowledgement

Special thanks to Dr Eyal Soreq for his expert opinions and patient guidance that fulfilled this rewarding experience.