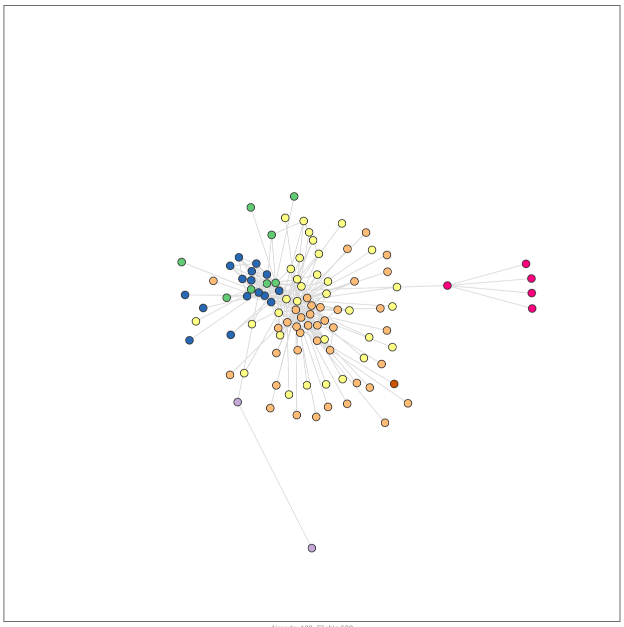
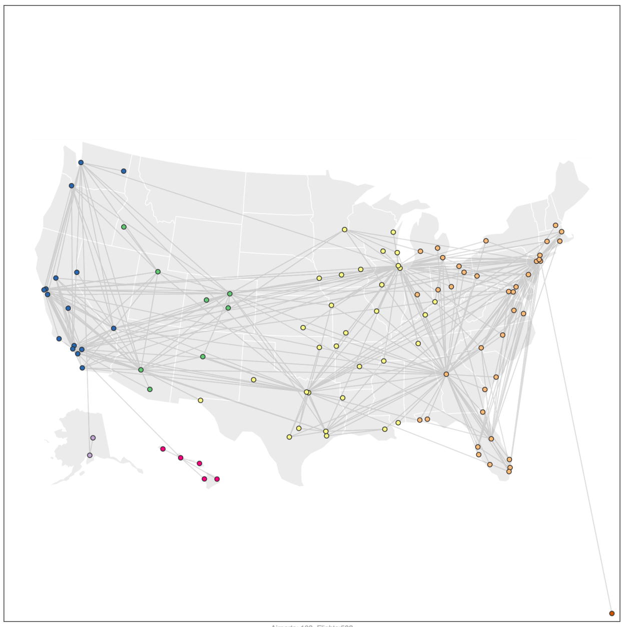
INF552 (2023-2024) - PC s07

Goal: visualize an air traffic network, as a force-directed node-link and as a geolocated network, using D3.

We will draw:


Airports: 103, Flights:582


Airports: 103, Flights:582

- US airports (color-coded by time zone) and their domestic connections (weight > 3000);
- and then the same network, but geolocated.

1. Data Structure

In function `loadData()`, fetch and parse the following files, using javascript Promises as in PC s06:

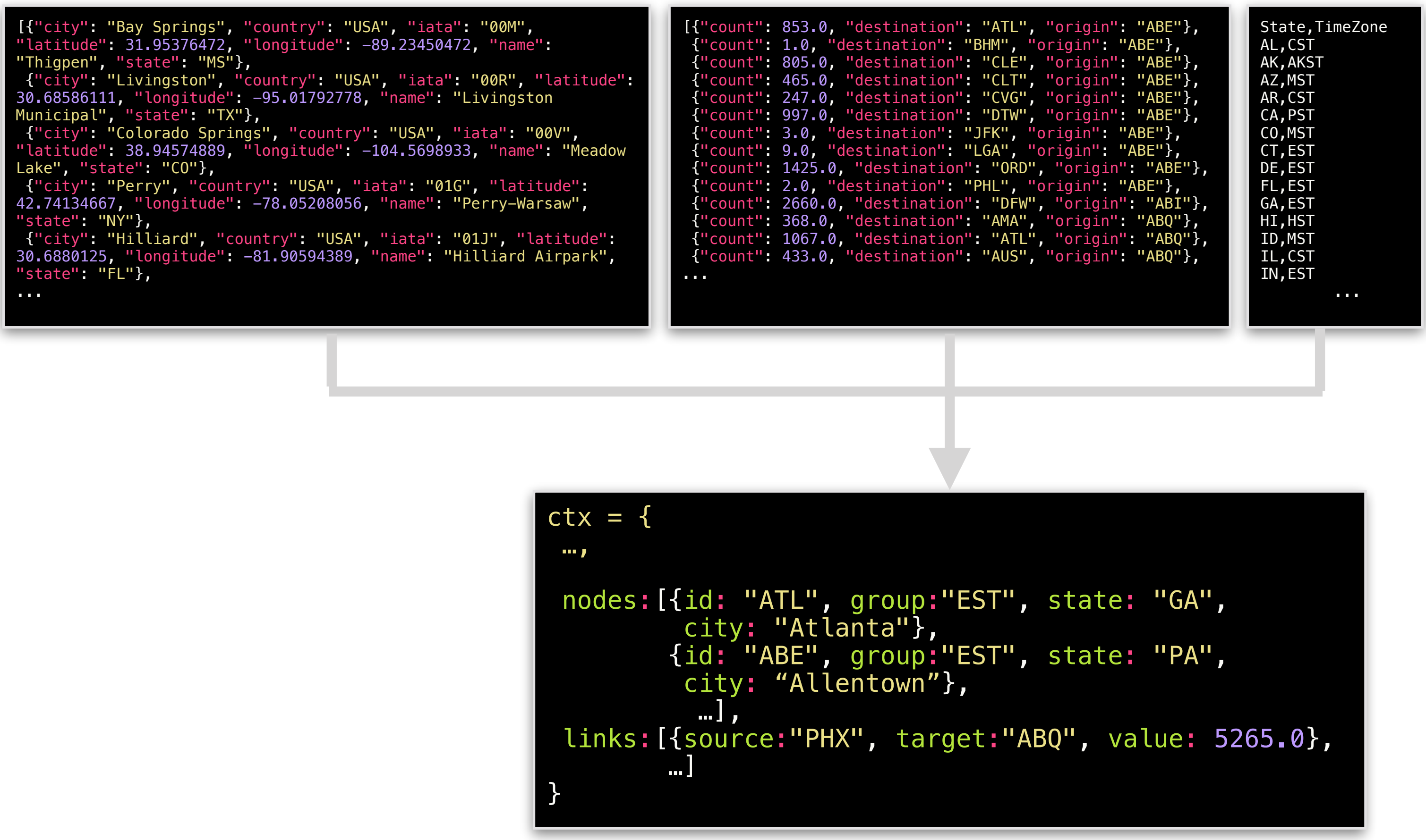
File	Description
`airports.json`	data about airports (IATA code, name, lat/lon, *etc.*)
`flights.json`	flights connecting those airports
`states_tz.csv`	time zones for each state
`us-states.geojson`	GeoJSON file for all 50 US states

Reminder: `Promise.all()` handles the multiple asynchronous calls to `d3.json()` and `d3.csv()`.

The network consists of airports (nodes) in the USA connected by flights (edges). It is hierarchical:

- each airport belongs to a state (`AL, MA, CA`, *etc.*);
- edges are weighted using a numerical attribute: `count`, representing the passenger traffic on each edge.

Transform the input data to match the data structure expected by d3's forceSimulation.



```
[{"city": "Bay Springs", "country": "USA", "iata": "00M",
"latitude": 31.95376472, "longitude": -89.23450472, "name":
"Thigpen", "state": "MS"},
 {"city": "Livingston", "country": "USA", "iata": "00R", "latitude":
30.68586111, "longitude": -95.01792778, "name": "Livingston
Municipal", "state": "TX"},
 {"city": "Colorado Springs", "country": "USA", "iata": "00V",
"latitude": 38.94574889, "longitude": -104.5698933, "name": "Meadow
Lake", "state": "CO"},
 {"city": "Perry", "country": "USA", "iata": "01G", "latitude":
42.74134667, "longitude": -78.05208056, "name": "Perry-Warsaw",
"state": "NY"},
 {"city": "Hilliard", "country": "USA", "iata": "01J", "latitude":
30.6880125, "longitude": -81.90594389, "name": "Hilliard Airpark",
"state": "FL"},
...
```

```
[{"count": 853.0, "destination": "ATL", "origin": "ABE"},
 {"count": 1.0, "destination": "BHM", "origin": "ABE"},
 {"count": 805.0, "destination": "CLE", "origin": "ABE"},
 {"count": 465.0, "destination": "CLT", "origin": "ABE"},
 {"count": 247.0, "destination": "CVG", "origin": "ABE"},
 {"count": 997.0, "destination": "DTW", "origin": "ABE"},
 {"count": 3.0, "destination": "JFK", "origin": "ABE"},
 {"count": 9.0, "destination": "LGA", "origin": "ABE"},
 {"count": 1425.0, "destination": "ORD", "origin": "ABE"},
 {"count": 2.0, "destination": "PHL", "origin": "ABE"},
 {"count": 2660.0, "destination": "DFW", "origin": "ABI"},
 {"count": 368.0, "destination": "AMA", "origin": "ABQ"},
 {"count": 1067.0, "destination": "ATL", "origin": "ABQ"},
 {"count": 433.0, "destination": "AUS", "origin": "ABQ"},
...
```

```
State,TimeZone
AL,CST
AK,AKST
AZ,MST
AR,CST
CA,PST
CO,MST
CT,EST
DE,EST
FL,EST
GA,EST
HI,HST
ID,MST
IL,CST
IN,EST
    ...
```

```
ctx = {
 …,

 nodes:[{id: "ATL", group:"EST", state: "GA",
         city: "Atlanta"},
        {id: "ABE", group:"EST", state: "PA",
         city: “Allentown”},
          …],
 links:[{source:"PHX", target:"ABQ", value: 5265.0},
        …]
}
```

Put the nodes in `ctx.nodes`, the links in `ctx.links`.

The graph is quite large and noisy. Filter the data given as input to the graph layout algorithm:

- ignore airports with an IATA code starting with a number, such as, *e.g.*, `06C`;
- ignore flights whose `count < 3000`;
- ignore airports that are not connected (possibly as a result of the above flight filtering).

Add a `group` attribute to nodes (airports), whose value is the parent state's time zone.

Use the traffic volume (`count`) as the edges' weight (attribute `value`).

Add the state and city to node properties, as we will also display this information on demand.

2. Force-directed Layout

We now populate the SVG canvas:

- append two groups to the `<svg>` element: `<g id="links">` and `<g id="nodes">`;
- and bind elements from the `nodes` and `links` arrays to graphical marks as follows:
  - using the classic construct `d3.selectAll(…).data(…).enter()…`, populate `g#links` with `<line>` elements mapped to items in the `ctx.links` array you constructed in Section 1.1, and set the opacity of `g#links` to `ctx.LINK_ALPHA`;
  - similarly, populate `g#nodes` with `<circle>` elements mapped to items in the `ctx.nodes` array, and set the radius of all circles to `5`.

Use the provided color scale named `color()` to fill nodes based on the time zone of the airport's state. Given a time zone as input, this function will consistently return the same color from a categorical color scheme, ensuring that all nodes from the same time zone have the same color:

```
// https://github.com/d3/d3-scale-chromatic
let color = d3.scaleOrdinal(d3.schemeAccent);

// usage
someColor = color(aStateCode);
```

We now use `d3-force` to visualize the network as a node-link diagram:
https://github.com/d3/d3-force/blob/master/README.md

The code initialising the layout algorithm is already provided in `ex07.js`:

```
let simulation = d3.forceSimulation()
                   .force("link", d3.forceLink().id(function(d) { return d.id; })
                                                .distance(5).strength(0.08))
                   .force("charge", d3.forceManyBody())
                   .force("center", d3.forceCenter(ctx.w / 2, ctx.h / 2));
```

Associate the previously-created nodes (circles) and links (lines) to this simulation:

```
// input data structure created earlier is in ctx.nodes + ctx.links:

simulation.nodes(ctx.nodes)
          .on("tick", simStep);

simulation.force("link")
          .links(ctx.links);

function simStep(){
    // code run at each iteration of the simulation
    // updating the position of nodes and links
    d3.selectAll("#links line").attr("x1", (d) => (d.source.x))
                               .attr("y1", (d) => (d.source.y))
                               .attr("x2", (d) => (d.target.x))
                               .attr("y2", (d) => (d.target.y));
    d3.selectAll("#nodes circle").attr("cx", (d) => (d.x))
                                 .attr("cy", (d) => (d.y));
}
```

Important: when setting the link array in the simulation as done above, the simulation actually tampers with the properties of the objects in the `ctx.links` array. The `source` and `target` attributes of all your links are no longer strings with the IATA code of the corresponding airport nodes (which you did set earlier). Now, they are references to those node objects themselves (from `ctx.nodes`). See https://github.com/d3/d3-force#link_links for more detail (reading is not mandatory to finish this exercise).[1]

Finally, append a title to each node's circle. That title should be a string made of the city and airport IATA code. It will be displayed when hovering the node with the mouse cursor. Take inspiration from http://bl.ocks.org/ilyabo/1339996

3. Geographical Layout (optional)

We now want to be able to toggle between the above node-link diagram and a representation of the same network on a map of the US.

Append a `<g id="map">` to the root `<svg>` element, drawn *below* `g#nodes` and `g#links`, and populate that new `g#map` with the GeoJSON features representing the US states (already loaded in Section 1.1). Do this by creating a `geoPath()` generator, configured with the Albers projection already provided in the code (`ALBERS_PROJ`). The map gets drawn the same way as in the previous session, binding the GeoJSON `features` to `<path>` elements whose attribute `d` is generated by a simple call to the `geoPath` generator. Set class `'state'` on those `<path>` elements.

Revisiting the code that builds the data structure written in (Section 1.1), for each airport node, add to its properties the `(x,y)` coordinates that correspond to where that node should be drawn on the map, based on its lat/lon coordinates. **Do not** call those attributes `x` and `y`, as these are two of the reserved values that the simulation tampers with.[1] You need to project those geographical coordinates using the same projection used to draw the map.
Hint: `ALBERS_PROJ` is actually a function, that takes an array `[lon,lat]` as input.

Warning: `ALBERS_PROJ` returns `null` for one of the airport nodes: SJU (Puerto Rico). Handle that particular case by assigning it coordinates in one corner of the map.

Important: at this point, you are only storing these coordinates in the data structure. You are not assigning them to the nodes' `<circle>` elements.

Set `g#map`'s opacity to `0`, as we show the node-link (NL) diagram by default, not the geolocated network.

We now have everything required to write the last function, that actually transitions between the NL diagram and the geolocated network. Write the contents of function `switchVis(boolean)`, which toggles between the NL diagram and the map each time users hit the `switch` button or key `T`.
A video is available on Moodle, illustrating the expected result.

- By default, we show the NL diagram. Handle branch `showMap == true` first:
  - fade out all links with an animation (change the opacity of the parent `g#links`); once faded out, update the `<line>`'s endpoint coordinates to the position that nodes will have on the map (no need for a smooth transition here); fade them back in;
  - in parallel, fade the map in (change the opacity of `g#map`);
  - in parallel, animate all nodes to their new position on the map (Albers projected coords stored earlier).

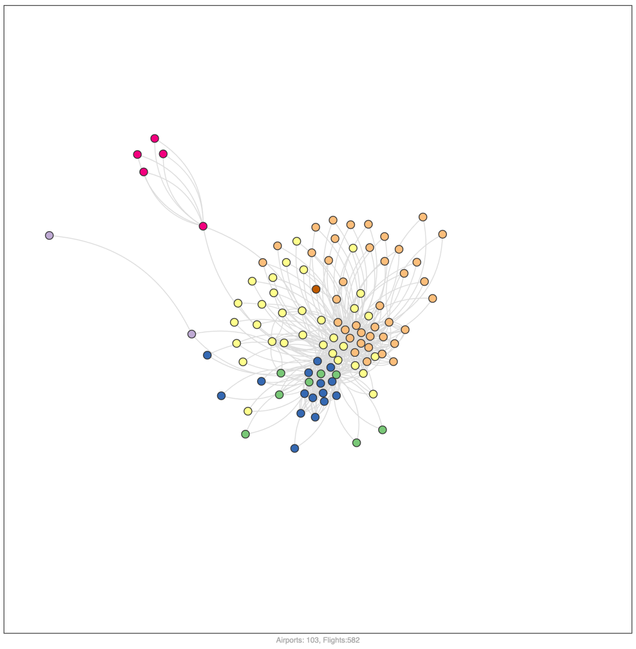
Important: you have to stop the force `simulation` *before* you do any of the above steps, otherwise nodes/links will be put back in place by the simulation.

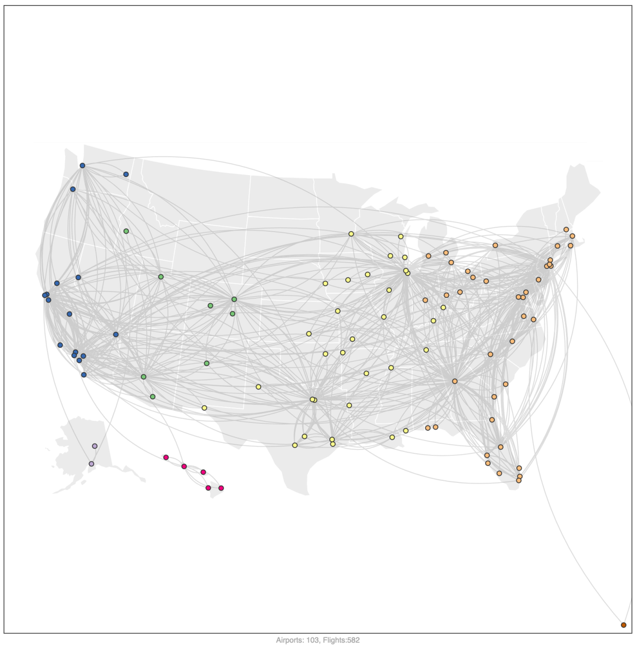
- We now handle branch `showMap == false`, which corresponds to the reverse transition, back to the NL diagram:
  - fade out the links, then update their endpoint coordinates to the position that nodes have in the NL diagram (stored in the link's `source` and `target` `(x,y)` attributes, see function `simStep()` on page 3 for reference); fade them back in;
  - in parallel, fade the map out;
  - in parallel, animate the nodes to these same positions.

[1] A similar thing happens to nodes. See https://github.com/d3/d3-force#simulation_nodes for more detail (again, reading not necessary to finish this exercise).

4. Curved Links (optional)

Replace `<line>` elements by `<path>` elements consisting of a single quadratic bézier curve:





This version reveals that we are actually dealing with a multi-graph.

Each `<path>`'s `d` attribute consists of:
- an `M` command to move (without drawing) to the source node's position;
- a `Q` command to draw a quadratic bézier curve from the above source node's position to the target node's position, with one control point controlling both tangents.

To achieve the same curvature as in the example above, the coordinates of that control point can be computed as:

$$x_{cp} = x_1 + \rho \cdot cos(\alpha + \frac{\pi}{6}) \qquad y_{cp} = y_1 + \rho \cdot sin(\alpha + \frac{\pi}{6})$$

where:

$$\rho = \frac{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}{2 \cdot cos(\frac{\pi}{6})}$$

$$\alpha = arctan2(\frac{y_2 - y_1}{x_2 - x_1})$$

(x_1, y_1) resp. (x_2, y_2) being the coordinates of the source resp. target nodes.

Reminder (`path/d` command syntax for `M` and `Q`):

```
Mx,y
Qcpx,cpy x,y
```

Example: `M10,10Q10,20 20,20`