Documentation

CSV Viewer with Graphs and Configurable Options

Description

A react app for viewing CSV data and visualizing graphs based on the data provided. Users can configure the graph to display specific data points, such as plotting over all the rows in a given column.

Features

- CSV Parser
- Scrollable tables to visualize the data
- Graphical visualization based on the data
- Possibility to select which data points to display on the graph
- Pause, restart, speed up, and slow down graph animation buttons
- Tooltip to better see the data

Major Components

Dashboard.js

- Main Page: shows all of the other components
- Includes a button for user to select a .csv file for the time data and a button for user to select a .csv file for the concentration data, display the parsed data in a scrollable table after uploaded

```
<h2 className='data-title'>Time Data</h2>
```

```
<CSVParser onDataParsed={onDataParsedTime}/>
```

After uploading the file, the onDataParsedTime function will serve as the callback function so the CSVParser component can pass the parsed data back to the Dashboard. The function will then format the data in a better way for usage.

```
const onDataParsed = (data) => {

   const col = [];

   data.map((d) => {

     col.push(Object.keys(d));

   })

   setNodes(col[0])

   setParsedData(data);

 }



 const onDataParsedTime = (data) => {

   const col = [];

   data.map((d) => {

     col.push(d);

   })

   setTime(col)

   setParsedDataTime(data);

 }
```

- Only after the parsedData and parsedDataTime states are set, the graph will then appear. The graph div contains the graph and some options for selecting specific data points from the graph to be visualized.
- The Graph component accepts both the time and concentration data, as well as column and row for specific selections

```
{parsedData && parsedDataTime &&

        <div className='landing-graph'>

          <div className='landing-graph-options'>

            <Dropdown data={time} onDataSelected={handleTimeSelected} type={'times'}/>

            <Dropdown data={nodes} onDataSelected={handleNodeSelected} type={'nodes'}/>

          </div>

            <div className='landing-graph-display'>

              <GraphColor dataset={parsedData} time={parsedDataTime} row={selectTime} column={selectNodes}/>

            </div>

        </div>

        }
```

CSVParser.js

- The CSVParser uses the papaparse library for parsing the .csv files

  Using the Papa.parse function, we can choose to display the data’s header if there are any, skip any empty lines, and parse values as numbers.

  After done parsing, we use the callback function passed by the parent component to send the data.

```
const parseCSV = (file) => {

    Papa.parse(file, {

      header: false, //true if our file has header row

      skipEmptyLines: true,

      dynamicTyping: true, // Parse values as numbers

      complete: (res) => {

        console.log(res.data)

        onDataParsed(res.data)

      }

    })

  }
```

GraphColor.js

- The GraphColor component is where the graph is being made. It uses the D3.js library to make the graph. It starts by taking dataset, time, row, column, and velocity as props. The dataset is the column's data and time is the time data. If the user specified a row and/or a column, then the data used for the graph will be different.
- The useEffect hook uses the props to set up the initial data, labels, dataset, and time data which will be used by the graph. If the user selected one column and one time, for example, then the data will only be for that column, and so on

```
 useEffect(()=>{

      if (dataset.length > 0 && time.length > 0) {

        let initLabels = [];



        //Passing the correct data, labes, and time based on the
given props
```

```
        if(column.length == 1 && row.length == 1){

            setInitData([dataset[row][column]])

            setInitTime(time[row])

            setData(dataset.map((val, i) => val[column]))

            initLabels.push(column);

        } else if (row.length == 1){

            setData(dataset)

            setInitTime(time[row])

            setInitData(dataset[row]);

            const col = [];

            dataset.map((d, i) => {

                col.push(Object.keys(d));

            })

            initLabels = col[0].map(Number)

        } else if (column.length == 1){

            setInitTime(time[0])

            setData(dataset.map((val, i) => [val[column]]))

            initLabels.push(column);

            setInitData([dataset[0][column]])

        } else{

            setData(dataset)

            setInitTime(time[0])

            setInitData(dataset[0])
            const col = [];

            dataset.map((d, i) => {

                col.push(Object.keys(d));

            })

            initLabels = col[0].map(Number)

        }



        setLabels(initLabels)

      }

    },[dataset, time, row, column])
```

- Then we begging the graph by using the drawChart() function, and deleting any previous graphs made (if any were made). We do this because D3 makes new graphs if the data change and we call drawChart again. We also set the animation to false in the beginning and only set it to true if all the times are selected

```
    useEffect(()=>{
        if (initData && labels) {
            setAnimationEnabled(false)

d3.select(concentrationGraphRef.current).select('svg').remove();
            setIter(0)
            drawChart();
        }
    },[initData, labels, initTime])
```

- The animation is done by setting setInterval functions over the specified speed. This is because we have control over the speed which makes speed up and slow down functionalities possible. This function call the updateChart function which will be the function animating the graph

```
//set up animation
```

```
    useEffect(() => {

        if (animationEnabled) {

            const barInterval = setInterval(() => {

                if (iter < data.length){

                    updateChart(data[iter], time[iter])

                    setIter(iter+1);

                }

                else {

                    clearInterval(barInterval);

                }

            }, (iter==0) ? 0 : animationSpeedRef.current);

            return () => clearInterval(barInterval);

        }

    }, [animationEnabled, iter])
```

- The drawChart function creates the graph. It stars by creating the chart are and adding x-axis and y-xis

```
const drawChart = () => {

        const xMinValue = d3.min(labels);

        const xMaxValue = d3.max(labels);



        // create chart area

        const svg = d3

            .select(concentrationGraphRef.current)
            .append('svg')

            .attr("class", "chart")

            .attr("viewBox", [0, 0,  width + margin.left + margin.right,
height + margin.top + margin.bottom])

            .attr("width",  width + margin.left + margin.right)

            .attr("height", height + margin.top + margin.bottom)

            .attr("style", "max-width: 100%; height: auto;")

            .append('g')

            .attr('transform', `translate(${margin.left},${margin.top})`);
```

- The bar is made by using the initData at first, which will then be updated later.

```
const bar = svg
        .append("g")
        .selectAll("rect")
        .data(initData)
        .join("rect")
        .attr("fill", d => colorScale(d))
        .join("rect")
        .style("mix-blend-mode", "multiply")
        .attr('class', 'bar')
        .attr("x", (d, i) => xScale(labels[i]) - barWidth / 2) //
center the bars on the tick marks
        .attr("y", height - 100)
        .attr("height", 100)
        .attr("width", barWidth)
        .text(function(d) { return d; })
        .on("mouseover", function(event, d){
            return tooltip.style("visibility", "visible");})
        .on("mousemove", function(event, d){
            tooltip.text(d);
```

```
                return tooltip.style("top",
  (event.pageY-10)+"px").style("left",(event.pageX+10)+"px");})
            .on("mouseout", function(){
                return tooltip.style("visibility", "hidden");})
```

- Lastly the updateChart function updates the graph by calling the transition function with the giving velocity

```
    const updateChart = (newData, time) => {

        let svg = d3.select(concentrationGraphRef.current)

        const bars = svg.selectAll(".bar")
          .data(newData)
          .text(function(d) {return d})

        svg.select('.chart-title').text('Time: '+ time + 's');

        bars
        .transition()
        .duration(animationSpeedRef.current/2)   // duration of the
transition in milliseconds
        .attr("fill", d => colorScale(d))
    };
```

Table.js

- The Table component displays a scrollable table using the Material UI and React Virtuoso libraries.

  It has data and a title props. The data will be used to populate the table and the title as the string for header texts

- The fixedHeaderContent function creates the column headers and leave them ‘stick’ so when the user scrolls down they still appear
- It checks for a ‘Time’ title so it can choose to not display the columns index in this case

```
function fixedHeaderContent() {

    return (

      <TableRow>

        {columns.map((column) => (

          <TableCell

            key={column.dataKey}

            variant="head"

            align={column.numeric || false ? 'right' : 'left'}

            style={{ width: column.width, backgroundColor: '#A3AFC8'
}}

            sx={{

              backgroundColor: 'background.paper',

            }}

          >

            {title} {title == 'Time (sec)' ? '' : column.label}

          </TableCell>

        ))}

      </TableRow>
    );

  }
```

- The rowContent function set the table’s rows using the column to map it by its index

```
function rowContent(_index, row) {

  return (

    <React.Fragment>

      {columns.map((column) => (

        <TableCell

          key={column.dataKey}

          align={column.numeric || false ? 'right' : 'left'}



        >

          {row[column.dataKey]}

        </TableCell>

      ))}

    </React.Fragment>

  );

}
```