PROPOSAL FOR GSoC 2023

Title: Network dependency visualization
Organization : Internet Health Report
Mentor: Romain Fontugne

Abstract

This project aims to create a Vue.js component that will display network dependencies in a more intuitive and interactive way. The proposed visualization will represent the data as a graph, with each node representing an Autonomous System and links representing dependencies. The graph will also be annotated with additional metrics, such as latency, and offer mechanisms to display changes over time. Additionally, this will implement a feature to visualize network dependencies.

Personal Information

Name	Jaydip Dey
Email	jaydipdey2807@gmail.com
TimeZone	Indian Standard Time (IST)
University	Jadavpur University, Kolkata, India
Major	Computer Science and Engineering
Skills	HTML, CSS, JS, ReactJS, VueJS, Node.JS, Express.JS, MongoDB
LinkedIn Profile	https://www.linkedin.com/in/jaydip-dey/
GitHub Profile	https://github.com/jaydip1235

About my achievements: I am currently an SDE intern at Mercor and a Full Stack development Instructor at Placewit. I have also been the finalist of Smart India Hackathon 2022 and been an SDE Intern at Optum. Beside that I have also contributed to HacktoberFest and in many open source projects. I have won more than 10 Hackathons (inter-college and national) and am a hardworker and a quick learner. To see more about my achievements click here.

Related Work on data visualization

I have worked earlier with visjs (a Javascript visualization library) to create an application of Dijkstra's shortest path algorithm. Project can be found here

Project goals

1. Develop an intuitive and interactive graph-based visualization tool for the IHR website to represent network inter-dependencies.
   a. Each node in the graph represents an Autonomous System.
   b. Each link between nodes represents a dependency between networks.
   c. Incorporate the ability to display changes in network dependencies over time using simple line charts.
2. Annotate the graph with additional network metrics reported by IHR, such as latency.
3. Integrate the functionality of the offline tool, country-as-hegemony-viz, into the IHR website to visualize network dependencies at a country level.
4. Ensure the visualization tool is user-friendly, with easy-to-understand visuals and navigation options.
5. Provide documentation and support for the developed tool, ensuring maintainability and extensibility for future updates and improvements.
6. Test the developed tool for accuracy, performance, and responsiveness on various devices and platforms to ensure a seamless user experience.

Weekly Schedule

Community Bonding Period	• Engage with the IHR community and mentors to understand the project requirements and goals. • Familiarize with the IHR platform, codebase, and datasets. • Study visualization libraries, such as Plotly and D3.js, and decide which one to use for the project.
Week 1-2	• Designing the network dependency graph visualization component. • Implementing the basic graph structure with nodes and links (dependencies) using the chosen visualization library.
Week 3-4	• Adding mechanisms to display changes in network dependencies over time, such as a slider or time range selector. • Implementing animations or transitions to show changes in the graph over time.

Week 5-6	● Annotating the graph with additional metrics, such as latency, and integrating external datasets. ● Ensuring that the graph is interactive, allowing users to explore the data and annotations.
Week 7-8	● Integrating the existing offline tool for country-level network dependency visualization into the IHR website. ● Ensuring the tool is consistent with the IHR website's design and functionality.
Week 9-10	● Thoroughly testing the new components and features for functionality and performance. ● Optimizing the code and bug fixing discovered during testing.
Week 11-12	● Creating documentation for the newly developed components and features. ● Updating the IHR website's user guide with information about the network dependency visualization. ● Finalizing the project for submission.

Periodic meetings, taking regular feedback from the mentors and work on the same will be a part of all the weeks.

HLD and LLD

High-Level Design (HLD):

- ***System Architecture:***
  a. Data Collection Module: Collects and processes data about dependencies, latency, and other relevant metrics.
  b. Data Storage Module: Stores processed data in a suitable format for efficient retrieval and updates.
  c. Graph Visualization Module: Generates intuitive graph representations of the dependencies and related metrics.
  d. Country Dependency Visualization Module: Displays network dependencies for a specific country using the offline tool.
  e. Web Interface: Provides user-friendly access to the IHR website and its visualization features.

- Components:
  a. Data Collection and Processing
    - i. Data sources (e.g., Japan national academic ISP, University of Tokyo)
    - ii. Data preprocessing (e.g., aggregating data, extracting features)

  b. Data Storage

i. Database system (e.g., SQL, NoSQL)
ii. Data schema (e.g., tables, indexes)

c. Graph Visualization

i. Graph library (e.g., D3.js, Graphviz)
ii. Visualization options (e.g., node size, color, tooltip)

d. Country Dependency Visualization

i. Integration of the offline tool (country-as-hegemony-viz)
ii. Visualization customization (e.g., country selection, date range)

e. Web Interface

i. Frontend framework (Vue)
ii. User authentication and access control
iii. Responsive design

Low-Level Design (LLD):

- **Data Collection Module:**
  a. Defining APIs and protocols for collecting data from various sources.
  b. Implementing data preprocessing algorithms to clean, validate, and structure data.

- **Data Storage Module:**
  a. Designing a database schema that efficiently represents dependencies and other metrics.
  b. Implementing data access layer for retrieving, updating, and deleting records.
- **Graph Visualization Module:**
  a. Selection of a suitable graph library and implement graph creation and rendering logic.
  b. Defining options for customizing visualization (e.g., node size, color, tooltip).
  c. Implementing methods for handling user interactions with the graph (e.g., zoom, pan).

- **Country Dependency Visualization Module:**
  a. Integrating the country-as-hegemony-viz tool into the system.
  b. Creating an API to serve country-specific dependency data to the visualization module.
  c. Implementation of user interface components for selecting countries and date ranges.

- **Web Interface:**
  a. Developing a responsive web interface using a frontend framework.
  b. Implementing user authentication and access control for secure access to the website.
  c. Creation of a user-friendly layout with intuitive navigation and clear call-to-actions.

Implementation Details

Creation a VueJS component that show the dependency graph

To create a **Vue.js** component that shows a dependency graph using **D3.js**, first we need to install the required dependencies.We need to run the following commands to install Vue.js, D3.js, and the vue-d3 library:

```
npm install -S vue
npm install -S d3
npm install -S vue-d3
```

Once the dependencies are installed, a new Vue.js component called (let's say DependencyGraph.vue) is created.

The **Explanation and the Code snippet** is given below:

Explanation: The code snippet below renders a force-directed graph visualization using the **D3.js** library. The component accepts an array of data as a required prop, which includes nodes and links representing the dependencies between nodes.

When the component is mounted, an SVG element is created with a specified width and height. A D3 force simulation is then set up using the provided nodes and links data. The force simulation applies forces like link, charge, and center to the nodes in order to create an interactive graph layout.

The code creates two groups of elements: one for links (lines) and another for nodes (circles). It sets their visual attributes such as stroke color, opacity, width, and fill color. A drag behavior is added to the nodes, allowing users to interact with the graph by dragging nodes around. The drag behavior also updates the simulation's forces when nodes are dragged.

The force simulation's "**tick**" event is used to continuously update the positions of nodes and links as the simulation progresses, ensuring that the graph layout is updated in real-time. The component returns the graph reference, which can be used by other components or for further manipulation.

DependencyGraph.vue

```vue
import * as d3 from 'd3';
import { onMounted, ref } from 'vue';
export default {
  name: 'DependencyGraph',

  props: {
    data: {
      type: Array,
      required: true,
    },
  },
  setup(props) {
    const graph = ref(null);

    onMounted(() => {
      const width = 800;
      const height = 600;

      const svg = d3
        .select(graph.value)
        .append('svg')
        .attr('width', width)
        .attr('height', height);

      const simulation = d3
        .forceSimulation(props.data.nodes)
        .force(
          'link',
          d3.forceLink(props.data.links).id((d) => d.id),
        )
        .force('charge', d3.forceManyBody())
        .force('center', d3.forceCenter(width / 2, height / 2));

      const link = svg
        .append('g')
        .attr('stroke', '#999')
        .attr('stroke-opacity', 0.6)
        .selectAll('line')
        .data(props.data.links)
        .join('line')
        .attr('stroke-width', (d) => Math.sqrt(d.value));

      const node = svg
        .append('g')
        .attr('stroke', '#fff')
        .attr('stroke-width', 1.5)
        .selectAll('circle')
        .data(props.data.nodes)
        .join('circle')
        .attr('r', 5)
        .attr('fill', '#69b3a2')
        .call(drag(simulation));

      node.append('title').text((d) => d.id);

      simulation.on('tick', () => {
        link
          .attr('x1', (d) => d.source.x)
          .attr('y1', (d) => d.source.y)
          .attr('x2', (d) => d.target.x)
          .attr('y2', (d) => d.target.y);

        node.attr('cx', (d) => d.x).attr('cy', (d) => d.y);
      });
    });

    function drag(simulation) {
      function dragstarted(event, d) {
        if (!event.active) simulation.alphaTarget(0.3).restart();
        d.fx = d.x;
        d.fy = d.y;
      }
      function dragged(event, d) {
        d.fx = event.x;
        d.fy = event.y;
      }
      function dragended(event, d) {
        if (!event.active) simulation.alphaTarget(0);
        d.fx = null;
        d.fy = null;
      }
      return d3
        .drag()
        .on('start', dragstarted)
        .on('drag', dragged)
        .on('end', dragended);
    }
    return { graph };
  },
};
```

Adding mechanisms to display changes over time

To display changes over time in a dependency graph built using D3.js, the following mechanisms can be incorporated:

- **Data preparation:**

First, it is to be ensured that the data includes a timestamp or any other way to represent time, like an index or version number. Preprocess the data may be required to include this information.

```
const data = [
  {
    timestamp: '2022-01-01',
    nodes: [/*  nodes */],
    links: [/* links */],
  },
  // ... more time-stamped data
]
```

- **Creating a time slider:**

Adding an input element with a range type to use as a time slider. We can customize the min, max, and step attributes to match the time range and granularity of data.

```
<input type="range" id="timeSlider" min="0" max="100" value="0" step="1">
```

- **Setting up the D3.js force simulation:**

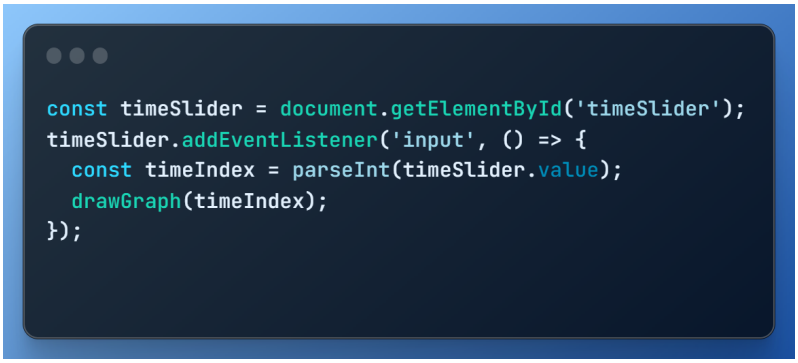
Creating a D3.js force simulation, including nodes and links, and drawing the initial state of the graph.

```
const simulation = d3.forceSimulation()
  .force('link', d3.forceLink().id(d => d.id))
  .force('charge', d3.forceManyBody())
  .force('center', d3.forceCenter(width / 2, height / 2));

function drawGraph(timeIndex) {
  // ... updating nodes and links based on data[timeIndex]
}
```

- **Updating the graph when the time slider changes:**

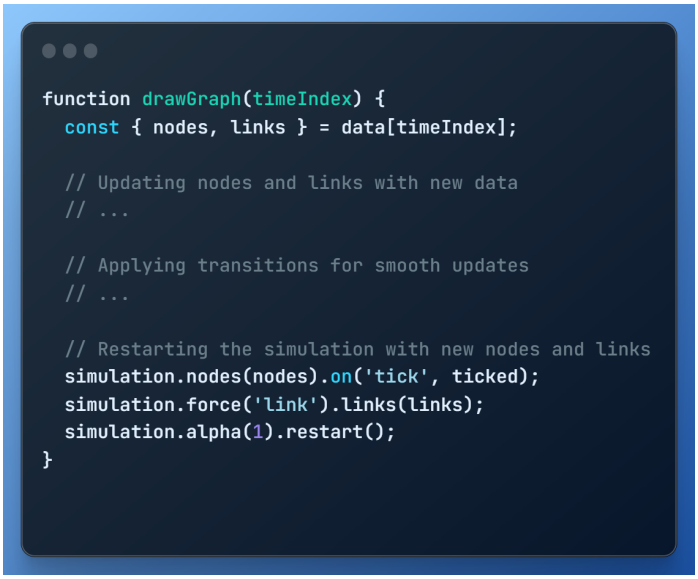
Listening for changes to the time slider, and updating the graph to display the appropriate state based on the slider's value.



```
const timeSlider = document.getElementById('timeSlider');
timeSlider.addEventListener('input', () => {
  const timeIndex = parseInt(timeSlider.value);
  drawGraph(timeIndex);
});
```

- **Implementing the drawGraph function:**

In the drawGraph function,we will update the nodes and links based on the data at the given time index.Transitions can be added to smooth the changes between graph states.



```
function drawGraph(timeIndex) {
  const { nodes, links } = data[timeIndex];

  // Updating nodes and links with new data
  // ...

  // Applying transitions for smooth updates
  // ...

  // Restarting the simulation with new nodes and links
  simulation.nodes(nodes).on('tick', ticked);
  simulation.force('link').links(links);
  simulation.alpha(1).restart();
}
```

By implementing these mechanisms, dependency graphs will be able to display changes over time. Users can interact with the time slider to see how the graph evolves. Styling and customizing graphs is needed to achieve desired visual representation.

Annotating the graph with other metrics and external datasets

To annotate a graph built with D3.js using other metrics and external datasets, these steps can be followed:

Load external datasets:

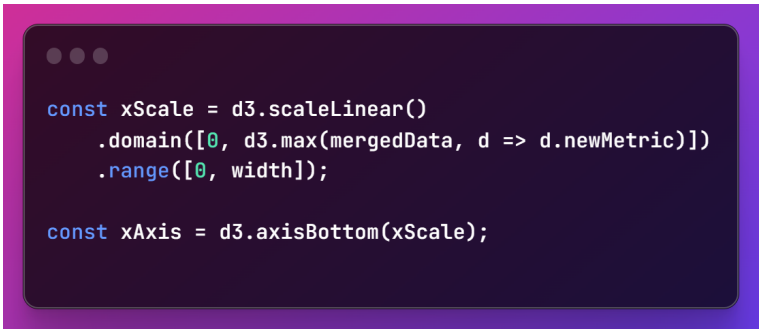
First, it is necessary to load the external datasets for annotation.We can use **d3.csv()**, **d3.json()**, or any other appropriate method to load data. Cleaning and preprocessing the data as needed.

```
d3.csv("path/to/your/csv.csv").then(function(data) {
    // data preprocessing here
});
```

Merge datasets:
If it is required to combine multiple datasets or metrics, we can use JavaScript array methods or a library like **Lodash** to merge them based on a common key.

```
const mergedData = data1.map(item => {
    const item2 = data2.find(d => d.key === item.key);
    return {...item, ...item2};
});
```

Create scales and axes:
We define appropriate scales and axes for new metrics. Depending on data, we might need to create linear, ordinal, or other scale types and update axes accordingly.



```
const xScale = d3.scaleLinear()
    .domain([0, d3.max(mergedData, d => d.newMetric)])
    .range([0, width]);

const xAxis = d3.axisBottom(xScale);
```

Update graph elements:
We need to add new SVG elements or modify existing ones to visualize the new metrics on the graph. Circles, bars, lines, or any other SVG element can be created depending on the requirements.

```
const circles = svg.selectAll(".circle")
    .data(mergedData)
    .enter()
    .append("circle")
    .attr("cx", d => xScale(d.newMetric))
    .attr("cy", d => yScale(d.existingMetric))
    .attr("r", 5)
    .attr("fill", "blue");
```

Add annotations:
To add annotations, the **d3-svg-annotation** library can be used. Loading the library, creating annotations using the appropriate options and finally appending them to SVG.

```
<script src="https://unpkg.com/d3-svg-annotation"></script>
// Create annotations
const annotations = [
    {
        type: d3.annotationLabel,
        note: {
            title: "Interesting Point",
            label: "This is an interesting data point",
            wrap: 200
        },
        x: xScale(interestingDataPoint.newMetric),
        y: yScale(interestingDataPoint.existingMetric),
        dy: -20,
        dx: 0
    }
];
// Create the annotation layer
const makeAnnotations = d3.annotation()
    .type(d3.annotationLabel)
    .annotations(annotations);

// Append the annotation layer to your SVG
svg.append("g")
    .attr("class", "annotation-group")
    .call(makeAnnotations);
```

By following these steps, we can annotate the **D3.js** graph with other metrics and external datasets.

Deliverables

Upon completion of the project, the IHR website will feature an enhanced interactive graph visualization and an integrated country-level network dependency analysis tool, providing network operators with a more intuitive and effective means to monitor the inter-dependence of networks and analyze network health.

Note of Thanks

I would like to thank my mentor Romain Fontugne for his extreme support and guidance towards writing this GSoC proposal. I am super excited to work on this project and with the Internet Health Report.

THANK YOU