

NetV.js: A Web-based Visualization Library for Large-scale Graphs

Dongming Han, Jiacheng Pan, Xiaodong Zhao^{a,b}, Wei Chen^{a,*}

^a*State Key Lab of CAD&CG, Zhejiang University, Hangzhou, Zhejiang, China*

^b*Zhejiang Lab, hangzhou, zhejiang, China*

Abstract

Graph visualization plays an important role in many fields, such as social media networks, protein-protein-interaction networks, and traffic networks. In the meantime, a dozen of visualization design tools and programming toolkits are widely used in graph-related applications. However, there remains a key challenge for efficiently visualizing large-scale graph data. In this paper, we present NetV.js, an open-source and WebGL-based Javascript library that supports fast visualization of large-scale graph data (up to 50 thousand nodes and 1 million edges). Experimental results demonstrate that our library outperforms existing toolkits (Sigma.js, D3.js, Cytoscape.js, and Stardust) on both the performance and memory consumption.

Keywords: Graph, Graph visualization, Node-Link diagrams, Web-based visualization.

1. Introduction

Graph visualization plays an important role in many fields, such as showing fraud transactions in financial data analysis [1], exploring information propagation in social media graph [2], and visualizing protein-protein-interaction in biological graph [3]. A series of visualization authoring tools [4, 5, 6] have been developed to facilitate visualization generation [7, 8, 9]. Notably, the proposal of d3.js [10] reduces the difficulty of web-based visualization authoring and enriches the visualization community. Thus, many graph visualization tools are also web-based. Node-link diagrams are widely used among many graph visualization solutions [11], because they reveal topology and connectivities [12]. Developers only need to control the nodes' positions and related links change their positions correspondingly. Node-link diagrams strongly rely on layout algorithms. In the meantime, interactions on nodes such as clicking and dragging are frequently needed. Developers need to pay special attention on these aspects using general visualization authoring tools such as d3.js [10], p4 [13, 14, 15], and stardust [16]. For example, using d3.js to create a node-link diagram needs to map data elements to graphical marks by means of the data-drive-documents scheme. The developer needs to directly handle visual elements and there is no hook/handler to handle the underlying graph. It may lead to bugs. Some graph visualization authoring tools such as Cytoscape.js [17] and Sigma.js [18] support graph visualization efficiently. Compared to general visualization tools, they encapsulate related interfaces of graph visualization, hide unrelated interfaces (e.g. controlling the position of a link) for developers, and expose some graph-related interfaces (e.g. accessing neighborhoods of a node). They improve the usability by leveraging the features of node-link diagrams.

For large-scale graphs, a large amount of graphical marks (e.g., nodes and edges) need to be processed. However, most tools have a limited capability in displaying graph data in real-time. According to our experiments in Section 5, a heavy delay occurs in visualizing graphs which have more than 5 thousand elements, leading unpleasing user experiences.

To the best of our knowledge, no existing tool can meet the requirements of developer usability and user accessibility at the same time. We analyze the design requirements of node-link diagram visualization, design and implement NetV.js, a web-based high-performance node-link diagram visualization library. It provides the developer usability by means of a suite of node-link related features and interfaces and increases user accessibility by utilizing the high-performance rendering ability of GPU. We evaluate the usability and accessibility of our implementation with comparative experiments. We contribute NetV.js, an open source library online (<https://netv.zjuvag.org/>).

2. Related Work

Many libraries [19, 20, 21] (tools, grammars, and frameworks) have been provided for implementing graph-based applications [22, 23]. At the beginning, developers use conventional programming languages to construct graph visualization applications, such as C Sharp, C++, Javascript, and Python. Developers need to have proficient programming skills and understand the implementation mechanism [24, 25]. In the meantime, they also spend a lot of time developing and debugging. To make it easier for developers to program and develop quickly, visualization grammars and frameworks [8, 9, 26] allows for detailed configurations on visual channels of visualizations such as D3.js [10], ECharts [19], Vega [27], and Vega-Lite [28].

*Corresponding author

Email address: chenvis@zju.edu.cn (Wei Chen)

65 However, when developers construct graph visualizations, they need to carefully select complex and different API to construct a graph, because these APIs are designed for full visualization rather than graph visualization applications. Cy-¹²⁰
 70 toscape.js [17], Sigma.js [18] and Gephi [29] are used to construct graph visualizations. They encapsulate a series of API which are elaborated for the graph data. Developers can use them to construct graph visualizations quickly. Moreover, large-¹²⁵
 75 scale data brings new challenges. Some libraries [30, 31, 32] use Canvas to render elements. However, these libraries can only render thousands of elements. To address this issue, Pix-¹³⁰
 80 iJS [33], P5 [14] and Stardust [16] use GPU-based acceleration technology to render a large number of elements. However, these GPU-based tools are also not designed for graph visualizations. They still have redundancy and complex API that have nothing to do with graph visualizations. It leads to a decrease in rendering efficiency and an increase in learning costs.¹³⁵

NetV.js targets on high-efficient visualization of large-scale node-link diagrams. It leverages a GPU-based visualization framework to improve the rendering performance, and design-friendly concise programming interfaces for efficient manipulation over graph elements.¹⁴⁰

3. Design

3.1. Design Requirements

To explore the design space of NetV.js, we interviewed three graph visualization experts, investigated five graph visualization tools including Gephi [29], Cytoscape.js [17], SNAP [34], Sigma.js [18], and GraphViZ [35]. We summarized the following design requirements for high-performance node-link dia-¹⁴⁵
 gram visualization:

R1 An abstract graph model to manipulate graph data.

In a node-link diagram, visual elements (graphical marks) are one-to-one corresponding to data elements (nodes and links). Developers only need to manipulate graph data elements rather than to access visual elements. To simplify the manipulation of origin data and visualization, developers need an abstract graph model. Several features should be supported:¹⁵⁰

R1.1 The node-link connection. The basic characteristic of graph is that each link connects two nodes. Developers only need to focus on modifying node position and ignoring the position of links when drawing a node-link diagram. The position of links should be automatically determined.¹⁵⁵

R1.2 The accessibility of neighbors. Another characteristic of graph is the connections among one node to its neighbor nodes and neighbor links. We found that many graph visualization systems support highlighting neighbor nodes and links of the focused node. All three experts agreed that the graph model needs to support a node to access its neighbors.¹⁶⁰

R1.3 Basic graph theory algorithms. Graph algorithms play important roles in some scenarios. For example, some important measurements (e.g., node degree, node centrality) can highlight POIs (points of interest), and shortest path finding algorithms can help developers trace the association between two nodes.

R2 High frame rate for large-scale node-link diagrams.

The frame rate can affect the user experience. The user interface will respond slowly or even stop responding with a low frame rate. Base on our interview with three experts, a rule of thumb for high-performance rendering is “*reaching 30 FPS (Frames Per Second) for rendering node-link diagrams with more than 100 thousand visual elements*”.

R3 Basic styles for nodes and links. Developers often need to encode information on node-link diagrams, such as encoding degree with node size. Although circular nodes and straight links are popular in large-scale node-link diagrams, there are also nodes and links with diverse shapes and styles. The most common node shapes include circles, rectangles, and triangles, and the most common link shapes include straight lines and curves.

R4 Layout algorithms. The dependence of node-link diagrams on layouts is self-evident. Thus, NetV.js should provide some basic layout algorithms. Because of the diversity of layout algorithms, NetV.js should provide interfaces for developers to customize their layout algorithms.

R5 Custom labels. Labels show basic information of nodes and links. Although drawing labels for all elements can lead to visual clutter in large-scale node-link diagrams, developers may adopt some strategies to select a part of elements to draw their labels. NetV.js should provide interfaces for developers to draw custom labels.

R6 Basic interactions. Based on our interview with experts and investigation on graph visualization authoring systems, we found that node-link diagrams’ interactions can be decomposed into interactions on nodes, links, and the canvas. Mouse events like mouseover, mousemove, mouseout, mouseup, and mousedown should be supported to construct more complex interactions such as dragging nodes. Selection interactions such as lasso should also be supported.

3.2. Design Details

To fulfill the proposed requirements, we designed and implemented NetV.js. NetV.js consists of three main parts: *Graph Model Manager*, *Rendering Engine*, and *Interaction Manager* (Figure 1). We isolate **R4**, **R5**, and selection interactions (**R6**) into three different plugins to reduce the code length of NetV.js.

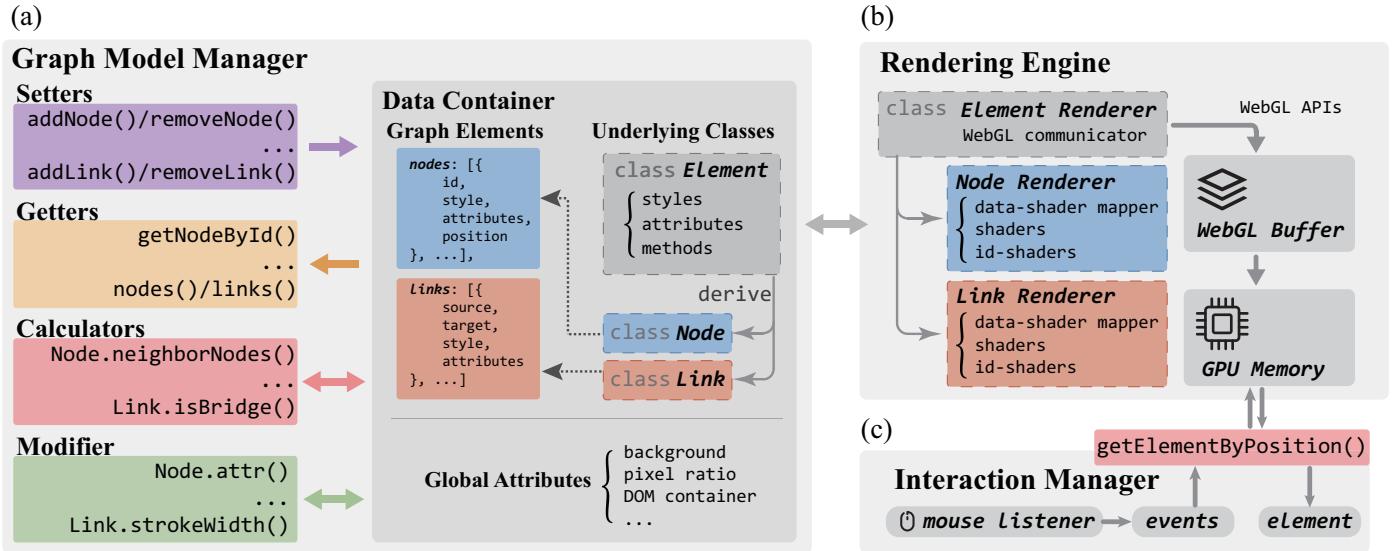


Figure 1: Our system design consists of three main parts: (a) *Graph Model Manager*, (b) *Rendering Engine*, and (c) *Interaction Manager*.

3.2.1. Graph Model Manager

To help developers manipulate data and visualization in a unified way (**R1**), we design *Graph Model Manager* in NetV.js (Figure 1(a)). The core of *Graph Model Manager* is an *Data Container*. It stores data elements, including nodes and links.²⁰⁵ Each node contains a unique identifier (**id**). Its styles are stored in **style** attribute and coordinates are stored in **position** attribute. Other information is stored in **attributes**. Similarly, each link contains a **source** and a **target** which represent two nodes the link connects. Styles and other information are stored in **style** and **attributes** in the same. In NetV.js, nodes and links are stored in objects instanced from classes **Node** and **Link**. They are derived from **Element**.

Adding, removing, modifying, query, and calculating data elements or visual elements are support. *Data Setters* are designed to add and remove data elements. *Data Getters* are designed to query elements (e.g. querying nodes by **ids**). Developers can modify the style and attributes of elements through *Data Modifier*. Several common graph algorithms are supported in NetV.js. Because they are not strongly related to graph visualization, we only implemented several interfaces such as neighbor nodes/links querying (**R1.2**) and shortest paths finding (**R1.3**). Besides, *Graph Model Manager* can also manipulate global configurations such as the canvas's background color and the DOM container.²²⁰

3.2.2. Rendering Engine

To fulfill **R2**, *Rendering Engine* utilizes the high-performance rendering ability of GPU. The workflow of *Rendering Engine* consists of 1) mapping styles and positions of elements into WebGL shader variables; 2) loading corresponding data into the WebGL buffer using WebGL APIs; 3) transferring data in the buffer into GPU memory and then drawing elements on the screen. In this way, *Rendering Engine* transfers data to the browser's CPU memory and the GPU memory in the end.²³⁰

We design a passive buffer modifying strategy: when developers modify styles of elements, *Graph Model Manager* will not call *Rendering Engine* to modify the WebGL buffer but stores the modified elements into a cache pool. When developers try to refresh the canvas, *Rendering Engine* will traverse elements in the cache pool to get related attributes and cover their corresponding buffer content. The relative strategy is to forwardly change the WebGL buffer when developers modify styles of elements. It is intuitive because the data flows forward. However, every time one element is modified, *Graph Model Manager* will call the related interface of *Rendering Engine* to change the buffer. It deepens the call stack and reduces the efficiency of rendering.

It is not allowed to modify the position of links directly (**R1.1**), but links will be cached into the pool when developers modify their connected nodes' position. *Rendering Engine* will update the WebGL buffer of links when the canvas is refreshed.

Rendering Engine supports rendering several basic styles of nodes and links (**R3**). Nodes can be rendered with four different shapes: circles, rectangles, triangles, and crosses. Developers can control the radius (**r**) of circular nodes, the **width** and the **height** of rectangular nodes, the vertices (**vertexAlpha**, **vertexBeta**, and **vertexGamma**) of triangular nodes, and the **thickness** and the **size** of cross nodes. Developers can set fill color (**fill**), the border color (**strokeColor**) and the border width (**strokeWidth**) of all kinds of nodes. Two different links are supported (the straight line and the curve). Developers can control the **curveness** of curve edges. Both two kinds of links support setting the color (**strokeColor**), the width (**strokeWidth**), and the dash array (**dashInterval**).

3.2.3. Interaction Manager

Interaction Manager provides a series of basic interactions (**R6**). NetV.js listens to several mouse events (mouseover, mouseout, mousemove, mousedown, mouseup and mousewheel) on

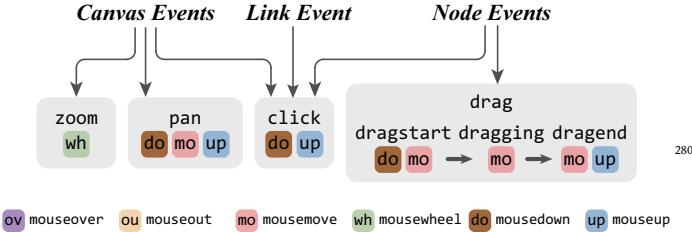


Figure 2: NetV.js interactions decomposition. Canvas events include zoom, pan, and click. Link event contains only click. Moreover, node events include click and drag.

280

the canvas. All these events are supported on nodes, links, and the canvas. We implemented several complex interactions by combining different events. For example, NetV.js supports the zoom+pan interaction by listening and combining mouse-²⁹⁰
wheel, mousedown, mousemove, and mouseup. Node dragging can be decomposed into several parts: dragstart, dragging, and dragend. When developers press the mouse and then move the mouse, dragstart will be triggered. Then dragging is triggered. When developers release the mouse, dragend and mouseup are triggered one by one. Figure 2 shows events on²⁹⁵
nodes, links, and the canvas.

To determine which element is hit by the mouse, *Rendering Engine* generates a hidden canvas that encodes the unique identifiers of elements into their corresponding pixels. Developers can use the interface **getElementByPosition()** to get the identifier encoded on the mouse-focused pixel so that the element on the mouse-focused pixel can be gathered by developers immediately.³⁰⁰

3.2.4. Layout Plugin

Visualizing a node-link diagram relies on a layout algorithm to give the relative position of nodes. Different layout algorithms have different interfaces; our plugin intends to design a set of unified interfaces to facilitate developer calling (R4):³⁰⁵

- **start()**: calling **start()** interface brings the algorithm into processing. To improve the frontend performance, NetV.js creates another thread (using asynchronous timer)³¹⁰ to calculate the layout result.
- **stop()**: calling **stop()** interface stops the layout thread immediately.³¹⁵
- **onTick()**: calling **onTick()** enables developers to visit the intermediate status of the layout algorithm with its callback. Some layout algorithms do not provide intermediate status. NetV.js generates intermediate status using interpolation functions. It makes the layout process more smooth.
- **onStop()**: calling **onStop()** raises the stopping status to developers and returns the layout result. It will be triggered when the algorithm finishes or developers call the **stop()** interface.³²⁰

To resolve R4, a radial tree layout and a force-directed layout based on these interfaces are implemented. In the future, we will increase more layout algorithms to facilitate visualization. Developers can also implement their layout algorithms using our interfaces.

3.2.5. Label Plugin

A label drawing plugin is implemented to help create labels for elements (R5). When drawing a large-scale node-link diagram, drawing labels for all elements will cause severe visual clutter. So, we assume that developers will only render labels for a few elements, for example, selecting some POI nodes or areas to draw labels. Thus, we chose SVG as label rendering tools to improve the expansibility. Besides the default textual label, developers can also customize labels by creating label templates.

3.2.6. Selection Plugin

A selection plugin is implemented to facilitate elements selection. Common selection interactions include the box selection and the lasso selection. Developers can get the selected structure (including nodes and links) in the callback function.

4. Examples

developers can use NetV.js to design graph visualization, explore and analyze graphs. In this section, we show diverse examples to illustrate the expressiveness and usability of NetV.js.

4.1. Basic Graph Drawing

Figure 4 shows a basic case. It can be regarded as three parts: initialization, loading data, and rendering. The ‘testData’ illustrates the graph data format. In this example, the initialization part hangs the canvas to the document ‘main’.

4.2. Customized Style

The most important function of graph visualization is to draw a graph with different styles, such as color, stroke, radius, and position of elements. Figure 3 shows the setting of customized style by using NetV.js. Developers can customize the style of each element and set the default style in the initialization part. In particular, initialize configuration items are set in the ‘configs’.

4.3. Interaction

A series of basic interactions are supported in NetV.js, including pan, zoom, mouseover, and so on. Figure 5 (b) shows several built-in interactions.

4.4. Plugin

NetV.js supports label rendering (Figure 3 (b)) with different drawing techniques such as SVG, Canvas, and WebGL. NetV.js also supports lasso interaction (Figure 3 (c)) to select nodes and different layout algorithms. Figure 5 (a) shows plugin configurations of the label, lasso, and layout.

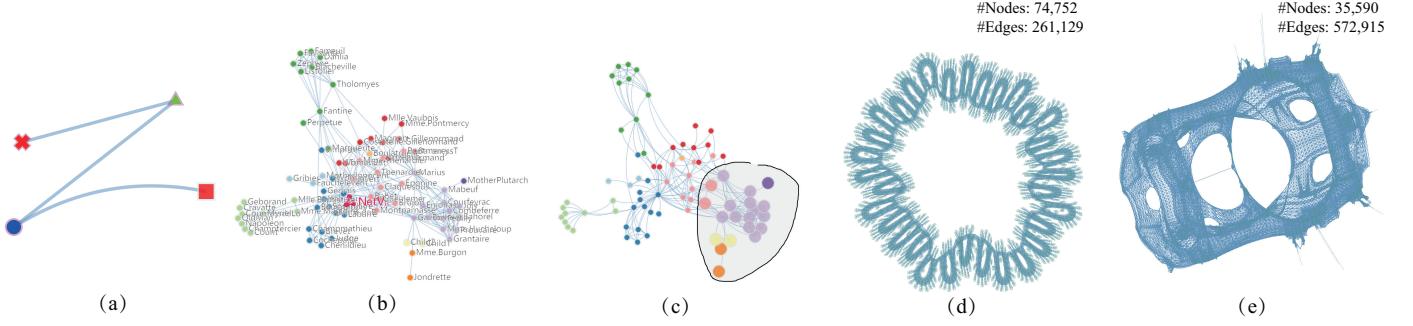


Figure 3: Illustrations of (a) the customized style, (b) the label plugin, (c) the lasso selection, (d) and (e) are two large-scale datasets.

```

const testData = {
  nodes: [
    {
      id: '0',
      x: 300,
      y: 100
    },
    {
      id: '1',
      x: 500,
      y: 100
    },
    {
      id: '2',
      x: 400,
      y: 400
    }
  ],
  links: [
    {
      source: '0',
      target: '2'
    },
    {
      source: '1',
      target: '2'
    }
  ]
}

const netv = new NetV({
  container: document.getElementById('main')
})
netv.data(testData)
netv.draw()

```

Figure 4: A basic unit of drawing three nodes and two edges.

```

const labelManager = new Label(netv)
labelManager.draw(netv.nodes(), (node) =>
  Label.template.rightText(node.id()))

const lasso = new Lasso(netv, {enable: true})
lasso.onSelected((selectedItems) => {
  netv.nodes().forEach((node) => {node.r(8)})
  selectedItems.forEach((node) => {node.r(16)})
  netv.draw()
})

const layout = new layouts RandomLayout(netv)
layout.time(1000)
layout.onStart(()=>{})
layout.onTick(()=>{})
layout.onStop(()=>{})
layout.start()

```

(a)

```

  netv.on('pan', () => {
    labelManager.updatePosition(netv.nodes())
  })
  netv.on('zoom', () => {
    labelManager.updatePosition(netv.nodes())
  })
  netv.nodes().forEach((node) =>
    node.on('dragging', () => {
      labelManager.updatePosition(node)
    })
  )

```

(b)

Figure 5: Code examples of (a) plugin configurations, and (b) build-in interactions.

4.5. Large-Scale Graph

NetV.js aims to render large-scale graphs. Figure 3(d) shows the graph visualization results of Finan512 dataset [36] with 74,752 nodes and 261,129 edges. Figure 3(e) shows the graph visualization results of bcsstk31 dataset [36] with 35,590 nodes and 572,915 edges. Thanks to WebGL, NetV.js can easily support millions of elements.

5. Experiment

We recorded the frame per second (FPS) experiment¹ to test the rendering performance of NetV.js with other popular tools and libraries which are supports graph rendering, including D3-SVG, D3-Canvas, Cytoscape.js, Sigma.js, and Stardust.js. In particular, NetV.js, Sigma.js, and Stardust.js use WebGL to render data. To simulate real-world graph data, we set the graph density as 20, which means the ratio of the number of edges to the number of nodes is 1 to 20. The display refresh rate is 144Hz; the GPU is GTX 1060 with 6G.

It is shown that Figure 6, NetV.js, Stardust.js, and D3-Canvas can render around 100 thousand elements. NetV.js can render more than 1 million elements with FPS greater than 1.

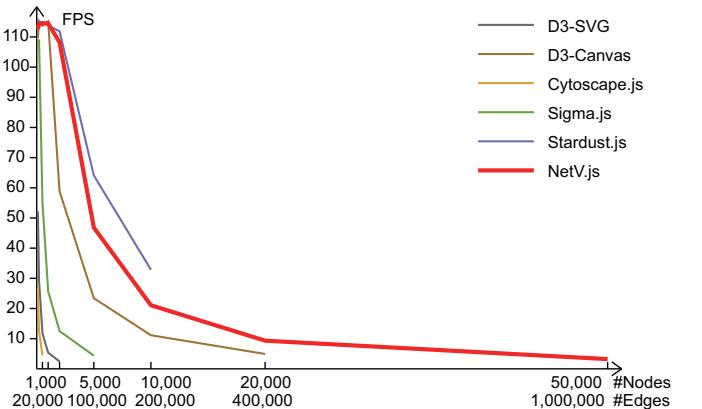


Figure 6: Performance comparison among 6 toolkits.

¹<https://github.com/ZJUVAG/NetV.js/tree/benchmarks/benchmarks>

6. Conclusion

This paper presents NetV.js, an open-sourced, WebGL-based Javascript library for large-scale node-link diagrams. In the future, we plan to extend NetV.js to support heterogeneous graphs. We also plan to explore more visualization components and visual analysis algorithms for analyzing graph data.

References

- [1] W. Chen, F. Guo, D. Han, J. Pan, X. Nie, J. Xia, X. Zhang, Structure-based suggestive exploration: A new approach for effective exploration of large networks, *IEEE Trans. Vis. Comput. Graph.* 25 (1) (2019) 555–565. doi: [10.1109/TVCG.2018.2865139](https://doi.org/10.1109/TVCG.2018.2865139)
- [2] M. A. Smith, B. Shneiderman, N. Milic-Frayling, E. M. Rodrigues, V. Barash, C. Dunne, T. Capone, A. Perer, E. Gleave, Analyzing (social media) networks with nodeXL, in: J. M. Carroll (Ed.), *Proceedings of the Fourth International Conference on Communities and Technologies, C&T 2009*, University Park, PA, USA, June 25–27, 2009, ACM, 2009, pp. 255–264. doi: [10.1145/1556460.1556497](https://doi.org/10.1145/1556460.1556497)
- [3] N. T. Doncheva, Y. Assenov, F. S. Domingues, M. Albrecht, Topological analysis and interactive visualization of biological networks and protein structures, *Nature protocols* 7 (4) (2012) 670.
- [4] A. Satyanarayan, K. Wongsuphasawat, J. Heer, Declarative interaction design for data visualization, in: *Proceedings of the 27th annual ACM symposium on User interface software and technology*, 2014, pp. 669–678.
- [5] G. G. Méndez, M. A. Nacenta, S. Vandenheste, ivolver: Interactive visual language for visualization extraction and reconstruction, in: *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, 2016, pp. 4073–4085.
- [6] N. W. Kim, E. Schweickart, Z. Liu, M. Dontcheva, W. Li, J. Popovic, H. Pfister, Data-driven guides: Supporting expressive design for information graphics, *IEEE transactions on visualization and computer graphics* 23 (1) (2016) 491–500.
- [7] J. Lu, J. Wang, H. Ye, Y. Gu, Z. Ding, M. Xu, W. Chen, Illustrating changes in time-series data with data video, *IEEE computer graphics and applications* 40 (2) (2020) 18–31.
- [8] A. Satyanarayan, J. Heer, Lyra: An interactive visualization design environment, in: *Computer Graphics Forum*, Vol. 33, Wiley Online Library, 2014, pp. 351–360.
- [9] J. Zong, D. Barnwal, R. Neogy, A. Satyanarayan, Lyra 2: Designing interactive visualizations by demonstration, *IEEE Transactions on Visualization and Computer Graphics*.
- [10] M. Bostock, V. Ogievetsky, J. Heer, D³ data-driven documents, *IEEE Trans. Vis. Comput. Graph.* 17 (12) (2011) 2301–2309. doi: [10.1109/TVCG.2011.185](https://doi.org/10.1109/TVCG.2011.185)
- [11] J.-c. Pan, D.-m. Han, F.-z. Guo, D.-W. Zhou, N. Cao, J.-r. He, M.-l. Xu, W. Chen, Rcanalyzer: visual analytics of rare categories in dynamic networks, *Frontiers of Information Technology & Electronic Engineering* 21 (2020) 491–506.
- [12] M. Ghoniem, J.-D. Fekete, P. Castagliola, A comparison of the readability of graphs using node-link and matrix-based representations, in: *Proceedings of IEEE Symposium on Information Visualization*, 2004, pp. 17–24. doi: [10.1109/INFVIS.2004.1](https://doi.org/10.1109/INFVIS.2004.1)
- [13] J. K. Li, K.-L. Ma, P4: Portable parallel processing pipelines for interactive information visualization, *IEEE transactions on visualization and computer graphics* 26 (3) (2018) 1548–1561.
- [14] J. K. Li, K. Ma, P5: portable progressive parallel processing pipelines for interactive data analysis and visualization, *IEEE Trans. Vis. Comput. Graph.* 26 (1) (2020) 1151–1160. doi: [10.1109/TVCG.2019.2934537](https://doi.org/10.1109/TVCG.2019.2934537)
- [15] J. K. Li, K.-L. Ma, P6: A declarative language for integrating machine learning in visual analytics, *IEEE Transactions on Visualization and Computer Graphics*.
- [16] D. Ren, B. Lee, T. Höllerer, Stardust: Accessible and transparent GPU support for information visualization rendering, *Comput. Graph. Forum* 36 (3) (2017) 179–188. doi: [10.1111/cgf.13178](https://doi.org/10.1111/cgf.13178)
- [17] M. Franz, C. T. Lopes, G. Huck, Y. Dong, S. O. Sümer, G. D. Bader, Cytoscape.js: a graph theory library for visualisation and analysis, *Bioinform.* 32 (2) (2016) 309–311. doi: [10.1093/bioinformatics/btv557](https://doi.org/10.1093/bioinformatics/btv557)
- [18] J. Coene, sigmajjs: An R htmlwidget interface to the sigma.js visualization library, *J. Open Source Softw.* 3 (28) (2018) 814. doi: [10.21105/joss.00814](https://doi.org/10.21105/joss.00814)
- [19] D. Li, H. Mei, Y. Shen, S. Su, W. Zhang, J. Wang, M. Zu, W. Chen, Echarts: A declarative framework for rapid construction of web-based visualization, *Vis. Informatics* 2 (2) (2018) 136–146. doi: [10.1016/j.visinf.2018.04.011](https://doi.org/10.1016/j.visinf.2018.04.011)
- [20] H. Mei, H. Guan, C. Xin, X. Wen, W. Chen, Datav: Data visualization on large high-resolution displays, *Visual Informatics* 4 (3) (2020) 12–23.
- [21] Tableau software, <https://www.tableau.com/> (Last Accessed: Jan. 2020).
- [22] X. Wang, W. Chen, J.-K. Chou, C. Bryan, H. Guan, W. Chen, R. Pan, K.-L. Ma, GraphProtector: A visual interface for employing and assessing multiple privacy preserving graph algorithms, *IEEE Transactions on Visualization and Computer Graphics* 25 (1) (2018) 193–203.
- [23] J. Pan, W. Chen, X. Zhao, S. Zhou, W. Zeng, M. Zhu, J. Chen, S. Fu, Y. Wu, Exemplar-based layout fine-tuning for node-link diagrams, *IEEE Transactions on Visualization and Computer Graphics*.
- [24] C. Reas, B. Fry, Processing: a learning environment for creating interactive web graphics, in: *ACM SIGGRAPH 2003 Web Graphics*, 2003, pp. 1–1.
- [25] C. Reas, B. Fry, Processing.org: a networked context for learning computer programming, in: *ACM SIGGRAPH 2005 web program*, 2005, pp. 14–es.
- [26] J. Heer, M. Bostock, Declarative language design for interactive visualization, *IEEE Transactions on Visualization and Computer Graphics* 16 (6) (2010) 1149–1156.
- [27] A. Satyanarayan, R. Russell, J. Hoffswell, J. Heer, Reactive vega: A streaming dataflow architecture for declarative interactive visualization, *IEEE Trans. Vis. Comput. Graph.* 22 (1) (2016) 659–668. doi: [10.1109/TVCG.2015.2467091](https://doi.org/10.1109/TVCG.2015.2467091)
- [28] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, J. Heer, Vega-lite: A grammar of interactive graphics, *IEEE Trans. Vis. Comput. Graph.* 23 (1) (2017) 341–350. doi: [10.1109/TVCG.2016.2599030](https://doi.org/10.1109/TVCG.2016.2599030)
- [29] M. Bastian, S. Heymann, M. Jacomy, Gephi: An open source software for exploring and manipulating networks, in: E. Adar, M. Hurst, T. Finin, N. S. Glance, N. Nicolov, B. L. Tseng (Eds.), *Proceedings of the Third International Conference on Weblogs and Social Media, ICWSM 2009*, San Jose, California, USA, May 17–20, 2009, The AAAI Press, 2009. URL: <http://aaai.org/ocs/index.php/ICWSM/09/paper/view/154>
- [30] J. Heer, S. K. Card, J. A. Landay, Prefuse: a toolkit for interactive information visualization, in: *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2005, pp. 421–430.
- [31] H. Wickham, ggplot2, *Wiley Interdisciplinary Reviews: Computational Statistics* 3 (2) (2011) 180–185.
- [32] D. Ren, T. Höllerer, X. Yuan, ivisdesigner: Expressive interactive design of information visualizations, *IEEE transactions on visualization and computer graphics* 20 (12) (2014) 2092–2101.
- [33] C. G. I. Graphics, Learn pixi.js.
- [34] J. Leskovec, R. Sosić, Snap: A general-purpose network analysis and graph-mining library, *ACM Transactions on Intelligent Systems and Technology (TIST)* 8 (1) (2016) 1.
- [35] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, G. Woodhull, Graphviz and dynagraph – static and dynamic graph drawing tools, in: *GRAPH DRAWING SOFTWARE*, Springer-Verlag, 2003, pp. 127–148.
- [36] T. A. Davis, Y. Hu, The university of florida sparse matrix collection, *ACM Transactions on Mathematical Software (TOMS)* 38 (1) (2011) 1–25.