ProtoGraph: A Non-Expert Toolkit for Creating Animated Graphs

Malchiel Rodrigues
Harvard University

Joel Dapello
Harvard University

Priyan Vaithilingam
Harvard University

Carolina Nobre° *
University of Toronto

Johanna Beyer° †
Harvard University

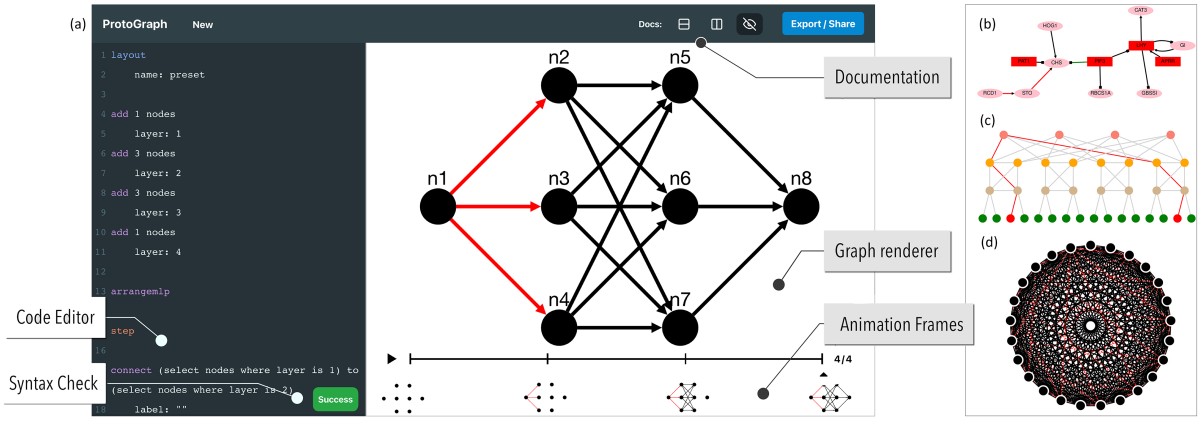

Figure 1: *ProtoGraph web tool interface* (a). The left column shows the code editor where the user can write ProtoGraph language statements, the right column displays the graph which dynamically renders as the user types in the editor. The bottom right panel shows the animation timeline previewing each step of the animation. The user can share the current graph using an auto-generated URL or export the visualization as an image or video. Sample graphs generated with ProtoGraph during the user study include a gene regulatory network (b), a FAT tree network (c), and a Mandala artwork using hundreds of edges (d).

Abstract

Creating intuitive and aesthetically pleasing visualizations and animations of small-to-moderate-sized graphs in the form of node-link diagrams is a common task across many fields, particularly in pedagogical settings. However, creating a graph visualization either requires users to manually construct a graph by hand or programming skills. We present ProtoGraph, an English-like programming language for non-expert users to rapidly specify and animate node-link graph visualizations. The language supports iterative prototyping, thereby allowing non-experts users to intuitively refine their graphs, and to easily create animated graphs. The key features of ProtoGraph include a web-based live coding interface, previews for the different states in an animated graph, integrated user documentation, and an active-learning style tutorial. We have integrated the ProtoGraph language into an open-source JavaScript graph visualization library for rendering and a graphical web interface for rapid prototyping. In a user study, we show that participants with varying coding experiences were able to quickly learn the ProtoGraph language and create real-world pedagogical visualizations, showing that ProtoGraph is easy to learn, efficient to use, and extensible.

Index Terms: Human-centered computing—Visualization—Visualization techniques—Graph drawings; Human-centered computing—Visualization—Visualization systems and tools—Visualization toolkits

1 Introduction

Visualizing network graphs is a deeply ingrained task in a variety of fields such as networking and software engineering, biology, neuroscience, and ecology, and often used for pedagogical purposes [3,6,18–20,29]. In addition to showing the structure of static graphs, visualizing dynamic graphs, where the structure or attributes change over time, is essential for a deeper understanding of these graphs and especially useful in pedagogical settings. For example, sequentially highlighting edges and nodes, or building the visualization one piece at a time can greatly facilitate understanding of the network.

Existing methods for specifying and animating graphs span a broad spectrum of complexity, from code-free drawing-based approaches to expressive programming-based tools. Drawing-based methods, such as PowerPoint, are effective for small graphs and short animation sequences, but quickly become time-consuming and inadequate for larger or more complex graphs. Furthermore, even small layout changes often require manual rearrangement of all edges and nodes. On the other hand, tools and programming languages that are designed for graph specification and animation (e.g., DOT [17]) are often powerful but present the user with a steep learning curve. As a result, they often require familiarity with programming and some degree of graph visualization knowledge [32].

Our work is inspired by the graph specification language DOT [17] and GraphVis [14]. However, we target non-expert users by focusing on learnability, iterative prototyping, and the easy and efficient creation of animated graphs. Our target user wants to create node-link diagrams quickly to sketch out ideas or demonstrate a graph-related concept. Example use cases include teaching biological pathways, or explaining the graph structure of neural nets. Users can construct nodes, edges, and attributes step-by-step or iteratively refine their graph (e.g., by later adding or modifying nodes and edges), similar to describing their graph verbally to a bystander, making it more intuitive for non-programmers.

In this paper, we present the *ProtoGraph* system, which includes a language, library, and web tool for increasing the ease and speed of rapidly prototyping static and dynamic graph visualizations. The **ProtoGraph language** offers an iterative graph specification paradigm, supports dynamically changing the graph structure and attributes for animating a graph, and is extensible. The

*e-mail: cnobre@cs.toronto.edu
†e-mail: jbeyer@g.harvard.edu
° indicates equal contribution

ProtoGraph library renders graph visualizations and animations specified in the format of the ProtoGraph language. Our **web-based graphical interface** integrates the ProtoGraph library to offer a live-coding environment. Users can navigate and preview the individual stages in an animated graph, and access integrated help and tutorials. Finally, we conducted a **user study** to evaluate the learnability, usability, and expressiveness of ProtoGraph. The study revealed that participants with varying programming experiences were able to quickly create their own pedagogical visualizations in ProtoGraph. ProtoGraph and its code are available at `https://protograph.io` and `https://github.com/protograph-io`, respectively.

2 Related Work

Graph Specification and Visualization. Graph specification refers to the way in which the structure and attributes of a graph are defined. Tools that support graph specification [2, 4, 7, 31, 35] often import standard graph file formats such as GEFX [7], GraphML [11], and DOT [17]. After importing graphs, tools such as GraphViz [14], Gephi [7], Cytoscape [38], GraphLet [22], GUESS [2], and Tulip [4] provide users with an interface to visualize the graphs. Changes to the graph can be made by updating the underlying graph structure and refreshing the visualization on demand.

Scripting-based solutions such as networkx [21], JGraph [5], Cytoscape.js [15], and others [10, 12, 13, 37], take a similar approach in that the graph is first specified – either externally or in the script itself – and then visualized on demand. This separation between graph definition and visualization can slow the process of iterating on graph structures. ProtoGraph addresses this by building the graph specification into the ProtoGraph language and providing a built-in code editor, which enables real-time visual feedback as the graph is being constructed. This immediate feedback loop between graph specification and visualization is critical to supporting quick iteration on graph prototypes. The benefit of reducing the barrier and delay between iterating and visualizing the specification in one interface has been shown in tools like edgy [8], Jupyter Notebooks [33], a Python library for Tulip [26], and Observable [9]. ProtoGraph continues this idea of rapid iteration, quick re-rendering, and integrated interfaces while allowing the user many of the benefits of traditional scripting and reducing the barrier to entry and complexity.

Graph Animation. Graph animation is the visual transformation of a graph between two states [16]. The key distinction between existing approaches is in how these states are defined. Defining animation states based on graph specification can be done by assigning time stamps to nodes and edges directly in the specifications of dynamic graphs [7], by sequencing static graphs [23], or by using a scripting approach to modify an existing graph structure into its new state [10, 12]. ProtoGraph simplifies the process of defining animation states by introducing *keyframes*. With a single keyword (*step*), users can define each state of the graph that should be animated. We describe keyframes as they are used for animation in more detail in Section 4.1. Animation is a key contribution of the ProtoGraph language, and the decision to incorporate state definition as a single keyword reflects the importance of this aspect. Additionally, our tool provides a playback interface that gives the user a visual overview of each state as well as control in navigating between them.

Expressive Visualization Authoring Tools Expressive visualization authoring tools provide design environments for generating visualizations that are comparable in expressiveness to programming-based tools [24, 25, 27, 34, 36]. These approaches have several similarities with the goals of ProtoGraph, including simplicity, ease of learning, and aimed at non-experts. However, the majority of these tools are aimed at generating standard visualizations and are not particularly geared towards graphs. Two exceptions are Graphies [34] and DataToon [24], which are designed for interactive graph authoring, and support some version of animation. Graphies for example supports

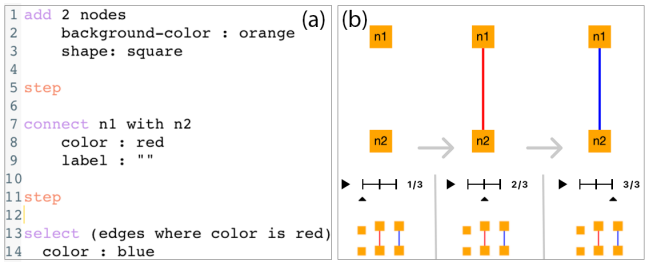

Figure 2: Using objects, commands, and selectors to create an animated graph. (a) ProtoGraph language source code. (b) Individual frames of the three-step animation sequence. For a live example of a more complex animation, visit `https://bit.ly/3zVJdE0`.

snapshots, which is similar in nature to ProtoGraph's *keyframes*. In contrast to ProtoGraph, however, they use direct manipulation, which limits scalability. ProtoGraph, on the other hand, uses a declarative language to easily define, iterate, and animate graphs.

3 Goals & Requirements

Our target users are people interested in quickly creating network graphs who are neither network nor graph visualization experts nor necessarily experienced programmers. Therefore, the goal of ProtoGraph is to provide a graph creation and visualization environment that is easy to learn and use, while being expressive and extensible.

We identified three technical and two user-focused requirements: **(R1) Graph structure specification** of nodes, edges, and associated data properties, such as edge direction or weight. Additionally, users must be able to specify visual graph properties, such as labels, visual styles, and layouts. **(R2) Graph animation** by dynamic changes to the graph structure, its data properties, or visual properties. **(R3) Real-time visual feedback** during graph specification and the ability to display dynamic graphs and their changes over time. **(R4) High learnability**. ProtoGraph must be easy to read and easy to write. We consider an integrated help system and onboarding learning material essential for learnability. **(R5) High efficiency**. The system must be efficient, enabling a high level of productivity [30], which we measure as the time needed to create visualizations.

4 ProtoGraph

The ProtoGraph system consists of three components: (1) the ProtoGraph language, for specifying graph structures, visual properties, and animation sequences; (2) the ProtoGraph library, an extensible parsing and rendering engine for the ProtoGraph language; and (3) the ProtoGraph webtool which embeds the ProtoGraph library into an interactive environment for users to create and share graph visualizations and animations online. Fig. 1(b-d) shows example graphs that were generated with the ProtoGraph web tool.

4.1 The ProtoGraph Language

We designed the ProtoGraph language with a declarative, English language-like syntax, with stylistic inspiration from the likes of YAML [39], SQL [28], and DOT [17], which are all widely regarded as approachable, intuitive, and easy to use and read (**R4, R5**).

We use three core semantic categories (objects, commands, and selectors) to define animated graphs and their properties (**R1, R2**).
Objects are stateful entities, such as nodes and edges, but also behind-the-scenes entities such as the `layout` object which defines global rendering properties.
Commands create, alter, or otherwise operate on objects. The command syntax is kept as simple and English-like as possible – a line starts with a named command, followed by any parameters taken by the command. Users can create new nodes using the `add` command, by specifying how many nodes are to be created. For instance,

`add 2 nodes` will generate two nodes with default visualization parameters and labels. When creating nodes, the user can immediately modify node attributes by indenting a set of key-value pairs following node creation, as shown in Fig. 2a (lines 1-3). Likewise, edges can be created and attributes modified with the `connect` command, as shown in Fig. 2a (lines 7-9). As a shorthand for generating nodes users can write `a1` to generate node a1, `a1 -> a2` to generate a directed edge between nodes a1 and a2, or chain the shorthand together: `a1 - a2 - a3` to create undirected edges between a1, a2, and a3. If nodes or edges already exist, the shorthand returns the attributes of the existing nodes or edges to be modified. A full list of commands can be found in our online documentation (`https://protograph.io/docs.html`).

The `step` command creates **animation** frames. Calling `step` creates a new frame that starts with the state of the prior frame. Users only have to specify the changes to the prior frame, as opposed to having to specify a complete graph in each frame. This structure makes specifying and rearranging animation frames particularly simple. Fig. 2 shows a simple animated graph with three frames.

Selectors query and return sets of nodes and/or edges. We chose an SQL-like selector system because of the English-like structure and popularity of SQL. A user can, for example, select all nodes with `select (nodes)`, or select edges with a given attribute (see Fig. 2a, lines 13-14). Parentheses around the query string to help resolve fundamental ambiguities in the English language, while also providing a clear visual grouping of what is being selected. Selectors are fully composable within commands operating on nodes or edges. To maintain English-like readability, when combining multiple selectors, we allow the user to drop the `select` keyword.

4.2 The ProtoGraph Library

To make ProtoGraph accessible to a wide range of users, we provide a TypeScript library for real-time parsing and rendering of the ProtoGraph language in web environments (**R3**).

The library includes a **parser** which validates and converts a string of ProtoGraph language input into JavaScript data objects. The **interpreter** is the orchestrator of the library; it takes the JavaScript data objects and directs the renderer to perform appropriate actions for the translated language fragment. The **renderer** stores the state of the graph and paints the final visualization. Our detailed documentation is available at `https://github.com/protograph-io`.

To ensure extensibility, the library accepts grammar fragments and logic handlers from extensions that extend the ProtoGraph language and system. New extensions can be implemented in either TypeScript or JavaScript. Parts of the ProtoGraph library, such as the SQL-like selector system, are implemented as extensions.

4.3 The ProtoGraph Web Tool

The ProtoGraph web tool (`https://protograph.io/`) provides an environment for users of varying experience levels to prototype graph visualizations in real-time on any device with a modern browser. The web tool offers a React.js interface and includes a web-based text editor with syntax highlighting and auto-complete (**R4, R5**), as well as a Render Pane with real-time visual feedback and playback controls and thumbnails of all animation steps (**R3**).

Code Editor Pane. The editor pane provides an interface for the user to define their graphs (Fig. 1a). The editor is based on the CodeMirror JS library [1] and provides syntax highlighting. It also offers auto-completion of commands and object keywords to show short descriptions and available options and parameters that the corresponding line supports. Finally, the editor in conjunction with the parser and interpreter provides error indicators with a line number, character highlighting, and a detailed error message.

Render Pane. By default, the real-time renderer displays the animation step of the graph that is currently being edited. For animated graphs, we further provide an overview of the entire animation sequence as small thumbnails in a navigation bar at the bottom of the render pane and provide media controls to playback the animation. In our current implementation, the render pane is built on top of Cytoscape.js [31], a JS graph theory and visualization library.

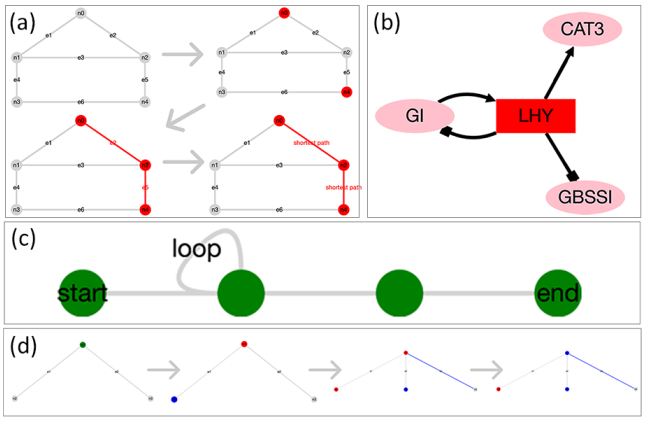

Figure 3: Example User Study Tasks. In Section I participants were asked to write code to best recreate a given graph visualization such as in (a) and (b). In Section II participants were asked to read the provided code and sketch out the corresponding graph on a digital whiteboard. Example answers are shown in (c) and (d).

Collaboration. We provide several features for collaboration among different devices and users. First, we auto-save a user's input locally, enabling them to leave and come back to their work. We can also store a user's input as URL parameters, enabling link sharing between users or devices. Finally, we support the export of graphs as PNG images or WebM videos.

Tutorial and Teaching Material. In the web tool's welcome screen, users can explore graph examples or quickly iterate on an idea by starting with a template that is related to what they envision. The ProtoGraph web tool also provides an active learning-based tutorial to help users quickly get started with the interface and language. The tutorial further shows users how to use the built-in documentation interface. This documentation interface provides users with a simple and easy way to learn about each part of the ProtoGraph Language and see relevant examples while using the tool.

5 Evaluation

We focus on evaluating the *learnability*, *usability*, and *expressiveness* of ProtoGraph. To that end, we conducted a crowdsourced user study to evaluate how ProtoGraph can be used by people of varying programming experiences. We measured users' ability to write and read ProtoGraph code and collected participants' subjective feedback. For details, we refer to the supplementary material.

Procedure. We recruited 41 participants on the crowdsourcing platform Prolific and paid them $15.00 USD for an estimated duration of 90 minutes. The study consisted of five phases: *Passive Training, Active Training, Trials, Study, and Demographics and Feedback*. The full study can be viewed at `http://protograph.projectalg.com/tool/study`.

Tasks. The study itself included three sections: writing with the ProtoGraph language, reading the ProtoGraph language, and a final free explore task. In the writing portion of the study, participants were given the image of a graph visualization or frames of a short animation and asked to recreate the visualization with ProtoGraph (Fig. 3 (a,b)). In the reading portion, participants were given an excerpt of code in the ProtoGraph language presented in a read-only version of the ProtoGraph internal code editor and asked to sketch the visualization with the provided digital whiteboard tool (Fig. 3

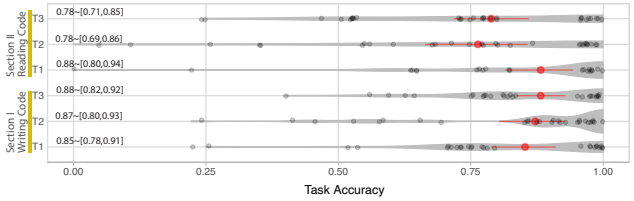

Figure 4: Participant accuracy on the three reading and three writing tasks in the study. On average, participants scored over 75% on all tasks. A red dot and line depict the average and 95% confidence interval, distribution shape is shown as light gray.

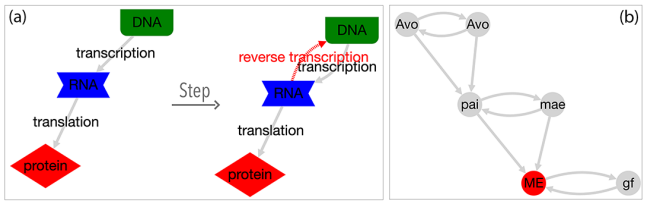

Figure 5: Results from two participants in Section III of the study, where we asked them to create their own visualizations. (a) Protein creation process from a participant with visualization experience of 5 out of 7 and coding experience of 5 out 7. (b) Family tree written in Portuguese (participant experience: vis 4/7, coding 1/7).

(c,d)). In the final free explore tool, we asked participants to create their own visualization in ProtoGraph.

Measures. We collected both quantitative and qualitative measures (i.e., completion times, tracked user interactions, submitted sketches and code, and subjective feedback using a 7-point Likert scale and free responses). To calculate correctness, two separate graders scored the individual components of each task separately (e.g., topology, layout, color, labels).

6 Results

We collected 39 valid study submissions (19 female, 19 male, 1 non-binary). Most participants had little or no coding experience (see Fig 6, left). We refer to the supplementary material for more detailed task descriptions, scoring methodology, and results. The study results are also available at `https://protograph.io/analysis/`.

Figure 4 shows the distribution of participant performance for the tasks in Sections I (writing) and II (reading) of the study. On average, participants scored over 75% on all tasks. Participants scored highest on tasks in the code writing portion, with averages between 80% and 90% accuracy. There was no significant impact of prior coding experience or experience with graph visualization on participants' accuracy, indicating that ProtoGraph is easy to adopt even by non-programmers and novices.

Section I: Writing. The writing section revealed that even with brief training and a short acquaintance period, participants were able to create graphs and adjust their visual properties successfully, validating the learnability of ProtoGraph (**R4**). Furthermore, as tasks progressed, participants were able to achieve the desired target with increasing accuracy. Task 2 required the use of a language extension, which participants were able to achieve with an average success rate over 75%; this validated the **extensibility** of ProtoGraph.

Section II: Reading. The ProtoGraph language syntax resulted in a high success rate for reading and determining topology, layout, and visual style of given code. Participants struggled most with reading label specifications (success rate of over 50%). This may have to do with ProtoGraph's optimization that allows names to be specified in the constructor or as a style attribute.

Section III: Free Explore. The optional Free Explore section allowed participants to create any graph visualization or animation. It validates that the ProtoGraph system can be used to solve real-world

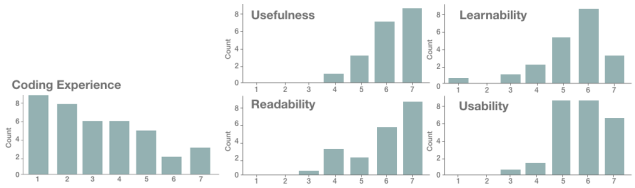

Figure 6: Left panel: Participant self-reported coding experience from 1 (no experience) to 7 (experienced coder). Right panel: Participant feedback on the usefulness, learnability, readability and usability of ProtoGraph from 1 (very difficult) to 7 (very easy).

scenarios Fig. 5 shows some notable examples. Fig. 5a, for instance, shows the multi-step protein creation process as an animation and fits the scenario where a biology teacher prepares a graphic for a slideshow. This participant also used features not introduced in the training, showing that they successfully navigated the documentation and autocompletion to find specific styles that they desired.

Participant Feedback. We collected user demographics and asked for feedback in a 7-point Likert scale and a free-responses. The vast majority of users rated ProtoGraph as highly useful, learnable, readable, and usable (Fig. 6). Especially the feedback from participants with self-reported coding experiences of 1 and 2 validate the approachability and ease of use for non-programmers, e.g., "For someone that has little experience with code it is simple to read it". A full report of user scores and feedback can be found in the supplementary material. Ultimately, the feedback shows that ProtoGraph was well received by participants at all levels of coding experience.

7 Discussion

The high success rate in which participants were able to complete the given tasks confirms ProtoGraph's ability to specify animated graphs and provide real-time visual feedback (**R1-R3**). Most of our participants had little programming experience and knowledge of graph visualizations. Yet, participants were able to complete tasks with high accuracy and increasing efficency (speed), using fewer documentation accesses as the study progressed. These results give us confidence in ProtoGraph's learnability (**R4**) and efficiency (**R5**).

Our study also revealed some limitations. First, participants only received minimal training (a 1.5-minute video, a short active training, and a trial to test comprehension). Participants might have performed better with a longer training session. Second, ProtoGraph is designed to present node-link diagrams in JavaScript/web-like environments, which limits its scalability. Thus, ProtoGraph is not well suited for large graphs (i.e., $> 5,000$ edges/elements or > 200 nodes sparse graph - depending on the layout algorithm).

8 Conclusion and Future Work

In this project, we set out to design a language and toolkit for creating graph visualizations and animations, specifically one that is easier to learn than current programmatic solutions and that is approachable for people with little to no programming experience. ProtoGraph can be useful for educators, collaborators, speakers, and anyone who creates small to medium-sized node-link diagrams or animations. Our integrated system allows for rapid prototyping with real-time feedback and minimizes memorability burdens and error rates while increasing efficiency by including syntax error reporting, autocomplete, and built-in documentation. Participants in our user study performed with high accuracy, showed impressive learning rates and increases in efficiency, and reported positive feedback.

Future work may explore questions about long-term learnability and efficiency. Further, research in natural language processing and artificial intelligence could provide a different interface to programming. With ProtoGraph we hope to inspire future research into making visualization systems more available and approachable to people outside of the visualization and computer science fields.

ACKNOWLEDGMENTS

This work was partly funded by NSF grant NCS-FO-2124179.

REFERENCES

[1] CodeMirror 5. https://codemirror.net/.

[2] E. Adar. GUESS: A language and interface for graph exploration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI '06*, p. 791. ACM Press. doi: 10.1145/1124772.1124889

[3] S. Asmuss and N. Budkina. On usage of visualization tools in teaching mathematics at universities. In *Proceedings of the 18th International Scientific Conference Engineering for Rural Development*, pp. 1962–1969, 05 2019. doi: 10.22616/ERDev2019.18.N515

[4] D. Auber, D. Archambault, R. Bourqui, M. Delest, J. Dubois, A. Lambert, P. Mary, M. Mathiaut, G. Melançon, B. Pinaud, B. Renoust, and J. Vallet. Tulip 5. In R. Alhajj and J. Rokne, eds., *Encyclopedia of Social Network Analysis and Mining*, pp. 1–28. Springer New York. doi: 10.1007/978-1-4614-7163-9_315-1

[5] J. Bagga and A. Heinz. Jgraph— a java based system for drawing graphs and running graph algorithms. In *Graph Drawing*, pp. 459–460. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. doi: 10.1007/3-540-45848-4_45

[6] J. E. Baker, I. F. Cruz, G. Liotta, and R. Tamassia. A new model for algorithm animation over the www. *ACM Computing Surveys*, 27(4):568–572, dec 1995. doi: 10.1145/234782.234792

[7] M. Bastian, S. Heymann, and M. Jacomy. Gephi: an open source software for exploring and manipulating networks. *International AAAI Conference on Weblogs and Social Media*, 8:361–362, 2009. doi: 10.13140/2.1.1341.1520

[8] S. Bird. Edgy. https://snapapps.github.io/.

[9] M. Bostock. Observable - Where teams explore, analyze, and communicate with data, together. https://observablehq.com/.

[10] M. Bostock, V. Ogievetsky, and J. Heer. D^3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011. doi: 10.1109/TVCG.2011.185

[11] U. Brandes, M. Eiglsperger, J. Lerner, and C. Pich. *Handbook of Graph Drawing and Visualization*, chap. Graph Markup Language (GraphML). CRC Press, 2014.

[12] J.-P. Coene. Sigmajs: An R htmlwidget interface to the sigma.js visualization library. *Journal of Open Source Software*, 3(28):814. doi: 10.21105/joss.00814

[13] G. Csardi and T. Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695.

[14] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull. Graphviz— Open Source Graph Drawing Tools. In *Graph Drawing*, Lecture Notes in Computer Science, pp. 483–484. Springer. doi: 10.1007/3-540-45848-4_57

[15] M. Franz, C. T. Lopes, G. Huck, Y. Dong, O. Sumer, and G. D. Bader. Cytoscape.js: A graph theory library for visualisation and analysis. *Bioinformatics*, 32(2):309–311. doi: 10.1093/bioinformatics/btv557

[16] C. Friedrich and P. Eades. Graph drawing in motion. vol. 6, 11 2002. doi: 10.1007/3-540-45848-4_18

[17] E. Gansner, E. Koutsofios, and S. North. Drawing graphs with dot. Technical report, 2006.

[18] S. Grissom, M. F. McNally, and T. Naps. Algorithm visualization in cs education: Comparing levels of student engagement. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, SoftVis '03, p. 87–94. Association for Computing Machinery, New York, NY, USA, 2003. doi: 10.1145/774833.774846

[19] P. J. Guo. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, p. 579–584. Association for Computing Machinery, New York, NY, USA, 2013. doi: 10.1145/2445196.2445368

[20] S. Hadjerrouit and H. H. Gautestad. Using the visualization tool simreal to orchestrate mathematical teaching for engineering students. In *IEEE Global Engineering Education Conference (EDUCON)*, pp. 38–42, 2018. doi: 10.1109/EDUCON.2018.8363206

[21] A. Hagberg, P. Swart, and D. S Chult. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference*.

[22] M. Himsolt. The graphlet system (system demonstration). In *Graph Drawing*, pp. 233–240. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[23] M. Jacobsson. D3-graphviz. https://github.com/magjac/d3-graphviz.

[24] N. W. Kim, N. Henry Riche, B. Bach, G. Xu, M. Brehmer, K. Hinckley, M. Pahud, H. Xia, M. J. McGuffin, and H. Pfister. DataToon: Drawing Dynamic Network Comics With Pen + Touch Interaction. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pp. 1–12. ACM. doi: 10.1145/3290605.3300335

[25] N. W. Kim, E. Schweickart, Z. Liu, M. Dontcheva, W. Li, J. Popovic, and H. Pfister. Data-Driven Guides: Supporting Expressive Design for Information Graphics. 23(1):491–500. doi: 10.1109/TVCG.2016.2598620

[26] A. Lambert and D. Auber. Graph analysis and visualization with Tulip-Python. In *EuroSciPy 2012 - 5th European meeting on Python in Science*. Bruxelles, Belgium, Aug. 2012.

[27] Z. Liu, J. Thompson, A. Wilson, M. Dontcheva, J. Delorey, S. Grigg, B. Kerr, and J. Stasko. Data Illustrator: Augmenting Vector Design Tools with Lazy Data Binding for Expressive Visualization Authoring. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pp. 1–13. ACM. doi: 10.1145/3173574.3173697

[28] J. Melton and A. R. Simon. *SQL: 1999: Understanding Relational Language Components*. Elsevier.

[29] T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. F. McNally, S. Rodger, and J. Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. In *ITiCSE-WGR '02*, 2002. doi: 10.1145/782941.782998

[30] J. Nielsen. *Usability Engineering*. Morgan Kaufmann, 1994.

[31] D. Otasek, J. H. Morris, J. Bouças, A. R. Pico, and B. Demchak. Cytoscape Automation: Empowering workflow-based network analysis. *Genome Biology*, 20(1):185, 2019. doi: 10.1186/s13059-019-1758-4

[32] K. Pantazos, S. Lauesen, and R. Vatrapu. End-user development of information visualization. In *End-User Development*, pp. 104–119. Springer, Berlin, Heidelberg, 2013.

[33] F. Perez and B. Granger. Project Jupyter: Computational narratives as the engine of collaborative data science, 2015. https://www.jupyter.org.

[34] H. Romat, C. Appert, and E. Pietriga. Expressive Authoring of Node-Link Diagrams With Graphies. *IEEE Transactions on Visualization and Computer Graphics*, 27(4):2329–2340, 2021. doi: 10.1109/TVCG.2019.2950932

[35] M. Santini. GraphvizAnim. https://github.com/mapio/GraphvizAnim.

[36] A. Satyanarayan and J. Heer. Lyra: An Interactive Visualization Design Environment. *Computer Graphics Forum*, 33(3):351–360, 2014. doi: 10.1111/cgf.12391

[37] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):341–350, 2017. doi: 10.1109/TVCG.2016.2599030

[38] P. Shannon, A. Markiel, O. Ozier, N. S. Baliga, J. T. Wang, D. Ramage, N. Amin, B. Schwikowski, and T. Ideker. Cytoscape: A software environment for integrated models of biomolecular interaction networks. *Genome research*, 13(11):2498–2504, 2003.

[39] V. Sinha, F. Doucet, C. Siska, R. Gupta, S. Liao, and A. Ghosh. YAML: A tool for hardware design visualization and capture. In *Proceedings 13th International Symposium on System Synthesis*, pp. 9–14, 2000. doi: 10.1109/ISSS.2000.874023