In-browser Visualization of Large-scale Graphs*

Luca Consalvi, Walter Didimo, Giuseppe Liotta and Fabrizio Montecchiani*

Dipartimento di Ingegneria, Università degli Studi di Perugia, Italy

Abstract

As today personal devices offer non-trivial amount of computing power and Web browsers provide powerful JavaScript engines, a recent stream of work focuses on building high-performance data analysis and management systems that run completely in the browser. Motivated by the fact that the use of visualization to present and analyze networks is taking a leading role in conveying information and knowledge to users that operate in multiple domains, the aim of our research is to explore the scalability limits of a system that executes the full graph visualization pipeline entirely in the browser.

In this paper, we present BrowVis, a self-contained system to compute interactive visualizations of large graphs in the browser. Experiments show that, on a common laptop, BrowVis can visualize graphs with thousands of elements in seconds, as well as graphs with millions of elements in minutes. Once the initial visualization has been computed, BrowVis makes it possible to interactively explore the represented graph by following a details-on-demand paradigm.

Keywords

Large-scale network visualization, visual analytics, in-browser computing

1. Introduction

Graphs appear as a natural model for representing data in various fields and application domains, such as for instance artificial intelligence [1], finance [2], recommender systems [3], social network analysis [4], and tourism [5]. In particular, the use of visualization to present and analyze networked data is taking a leading role in conveying information and knowledge to users that operate in the above mentioned domains. Indeed, when dealing with large graphs, a recent extended survey [6] revealed that visualization is a very popular and central task in graph processing pipelines. On the other hand, designing algorithms to produce valuable visualizations of large graphs is difficult, and it is reported in [6] as one of the most pressing graph processing challenges.

The problem of producing graph visualizations can be decomposed into a two-step graph processing pipeline, with iterations that might be triggered by user interaction. In the first step, a layout of the input graph is computed, which involves the assignment of geometric

ITADATA2022: The 1st Italian Conference on Big Data and Data Science, September 20–21, 2022, Milan, Italy

*This work is partially supported by the following projects: (*i*) MUR, grant 20174LF3T8 "AHeAD: efficient Algorithms for HArnessing networked Data"; (*ii*) MUR Project "Smartour" SCN_166; (*iii*) Department of Engineering - University of Perugia, grants RICBA20EDG and RICBA21LG.

*Corresponding author.

luca.consalvi@studenti.unipg.i (L. Consalvi); walter.didimo@unipg.it (W. Didimo); giuseppe.liotta@unipg.it (G. Liotta); fabrizio.montecchiani@unipg.it (F. Montecchiani)

0000-0002-4379-6059 (W. Didimo); 0000-0002-2886-9694 (G. Liotta); 0000-0002-0877-7063 (F. Montecchiani)

representations to the vertices and to the edges of the graph. In the second step, the layout is rendered on the screen through a user interface, which often enables the exploration of the displayed data via interaction primitives. The layout step involves the design of efficient algorithms to optimize some aesthetic criteria while satisfying given conventions and constraints. For instance, in the popular node-link paradigm, vertices are represented as points, edges are straight-line segments, and the layout should both avoid the clutter given by overlapping features and highlight possible symmetries in the underlying graph. The vast majority of the algorithms used to produce node-link layouts of large graphs follows force-directed methods [7, 8, 9], which usually yield super-linear time complexities. In fact, for the sake of efficiency, layout algorithms can be run remotely on powerful servers or cloud computing infrastructures (see, e.g., [10, 11]). The rendering step requires a careful use of the graphics system on the client side to avoid inefficiencies and artifacts, as well as the design of suitable interaction paradigms to support an effective exploration of the conveyed data. It is worth mentioning that interactive rendering paradigms may involve the use of visual abstractions of the input layout, aimed to reduce both the overall clutter of the visualization and the computational cost of displaying a huge amount of geometric elements (see, e.g., [12]).

Today personal devices offer non-trivial amount of computing power and Web browsers provide powerful JavaScript engines. As a consequence, a recent stream of work focuses on building high-performance data analysis and management systems that run completely in the browser. A limited list of examples include the following: El Gebaly and Lin [13] present an analytical relational DBMS implemented in JavaScript that runs within the browser; Lin [14] describes a self-contained JavaScript-based search engine; Lee et al. [15] propose a JavaScript implementation of a keyword spotting system that can be deployed directly on user devices. Also, recent platforms from the data profiling research area rely on Javascript engines and are capable of continuously updating visual components while maintaining high performances [16, 17, 18].

In this paper, we aim at exploring the scalability limits of a system that executes the entire graph visualization pipeline relying only on the JavaScript processing engine of the browser. The main motivation of our work is twofold: on the one hand, having the whole visualization produced in the browser cuts off network latency, enables offline usage, avoids security and privacy issues related to the transit and remote storage of the data, and removes any external dependency. On the other hand, according to very recent experiments [19], modern Web technologies for graph visualization struggle to achieve satisfactory performance already with graphs having up to few hundred thousand elements. Our main contribution is as follows:

- We describe BrowVis, a self-contained system to compute interactive visualizations of large graphs in the browser (Section 3). BrowVis significantly differs from existing Web technologies as it combines two best-in-class techniques to carry out the entire graph processing pipeline. Namely, a layout of the input graph is computed with a JavaScript porting of FM^3 [20, 21], a multi-level force-directed algorithm originally developed in C++. To perform rendering and interaction, BrowVis incorporates and adapts the main ideas behind LaGO [12], an OpenGL-based implementation of a technique to interactively render massive node-link layouts with adjustable level of abstraction.
- We provide a publicly available proof-of-concept implementation of BrowVis, and we report the outcome of an extensive experimental analysis aimed at assessing its per-

formance (Section 4). The experiments show that, on a common laptop, BrowVis can visualize graphs with several thousand elements in seconds, as well as graphs with millions of elements in minutes. Moreover, once the initial visualization has been computed, BrowVis makes it possible to interactively explore the represented graph by following a details-on-demand paradigm.

2. Background and Related Work

Below we provide some background on the key ingredients of our system, namely force-directed layout algorithms and rendering techniques, along with a brief overview of the related literature.

2.1. Force-directed layout algorithms

Force-directed algorithms are the most common solution to the problem of computing node-link layouts of general unrestricted graphs. They follow two basic principles: (*i*) edges should not be too long and hence adjacent vertices should be drawn near to each other; (*ii*) vertices should not overlap and should evenly distribute on the drawing area. These two principles can be encoded in a system of forces acting on the vertices of the input graph. We point the reader to the extensive surveys of Cheong and Si [7] and by Kobourov [9] on the vast literature on force-directed algorithms, as well as to the surveys by Hu and Shi [8] and by Landesberger et al. [22] that are focused on the visualization of large graphs. We also remark that the versatility of force-directed algorithms makes them suitable to visualize dynamic graphs [23], as well as clustered and compound graphs [24, 25]. Common to all force-directed algorithms are a model of the system of forces acting on the vertices and an iterative algorithm to find a static equilibrium of this system, which represents the final layout of the graph. In terms of running time, the main bottleneck lies in the fact that each vertex interacts with all other vertices, giving rise to an overall quadratic number of forces in each iteration of the algorithm. To alleviate this problem, different authors proposed spatial decomposition techniques to approximate forces acting between vertices that are far from each other [26, 27]. A quantum leap towards the applicability of force-directed algorithms to larger graphs is represented by *multilevel force-directed algorithms*, introduced in [28, 27, 29]. Algorithms in this family proceed along a framework that roughly works as follows: first the input graph is iteratively simplified via coarsening techniques, giving rise to a stack of coarser graphs; second, such a stack is traversed backward and a final layout of the original graph is obtained by progressively computing a layout for each intermediate graph in the sequence. As experimentally observed, FM^3 [30] is one of the most effective multilevel force-directed algorithms, as it produces less edge crossings and fewer vertex overlaps [31, 21].

In order to unleash the power of modern computing infrastructures, different implementation choices have been investigated; we briefly describe a very restricted list of examples. The first attempts to scale force-directed algorithms to very large graphs exploit the power of GPUs (see, e.g., [32]). They can draw graphs with a few million edges, but their development requires a low-level implementation tied to the computing platform. Parallel and distributed approaches have also been considered. For instance, Meyerhenke et al. [33] present a C++ implementation based on OpenMP of a layout algorithm using the maxent-stress metric for the layout optimization.

More recently, a series of works has pursued the use of modern Big Data frameworks such as Spark [34] and Giraph [10, 11].

2.2. Rendering techniques

The goal of rendering techniques is to avoid clutter and over-plotting, which are undesirable effects both in terms of readability and efficiency. Clutter occurs when many vertices and edges of the graph are drawn in small portions of the screen, giving rise to ambiguous blobs of pixels. This issue is unavoidable if we insist on drawing each single vertex and edge of a large graph containing more elements than the pixels the screen can offer. Moreover, dense portions of the graph force many edges to traverse common areas of the screen which, in turn, gives rise to plotting over the same pixels multiple times, a severe problem in terms of efficiency. Inspired by Shneiderman's mantra [35] "Overview first, zoom and filter, then details-on-demand", several authors proposed multilevel visualizations aimed at computing multiple abstractions of the input graph (see, e.g., [36, 37, 27, 38]). While seminal approaches in this direction bundle together layout and rendering, Zinsmaier et al. [12] propose an interactive rendering technique with adjustable levels of detail that operates directly on a given layout and does not require precomputed hierarchies or meshes. The approach in [12] consists of a combination of edge cumulation with density-based vertex aggregation, and its implementation exploits graphics hardware for speeding-up the computation. In the same spirit, Perrot and Auber [39] describe a multilevel system that works for any given layout in input. The main differences with respect to [12] are the use of different algorithms for vertex and edge aggregation and an implementation based on distributed platforms for Big Data processing.

While all above papers provide fundamental scientific groundwork for our research, none of the above techniques is conceived to run entirely in the browser. On the other hand, there exist many JavaScript-based libraries that can deal with both the layout and the rendering steps. In particular, Han et al. [19] present NetV.js, a JavaScript library for the visualization of large graphs, and compare its performance with several other JavaScript libraries for graph visualization, namely Cytoscape.js [40], D3.js [41], Sigma.js [42], and Stardust.js [43]. Based on their experiments, the authors conclude that Stardust.js and D3.js can render up to a total of one hundred thousand elements (both vertices and edges), while NetV.js can render up to a total of one million elements showing at least one frame per second. The main drawback of the experimental analysis in [19] is that no performance metric is reported (e.g., runtime or memory footprint) other than the framerate. Moreover, and most importantly, none of the above libraries provide abstractions but instead draw each single vertex and edge of the graph, thus incurring into both clutter and over-plotting.

3. The BrowVis System

In this section we describe the design of BrowVis, which embraces the concept of multilevel visualization in order to achieve both readability and scalability. The source code is publicly available[1]. Fig. 1 illustrate two graphs with different structures and sizes visualized with BrowVis.

[1] https://github.com/Luk4e/graph_visualization

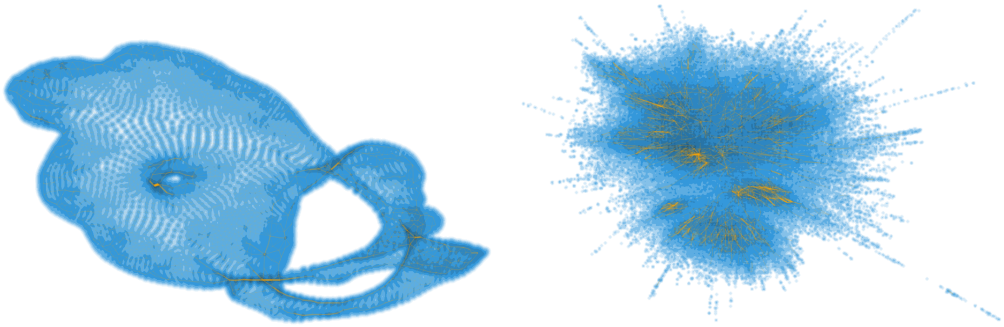

Figure 1: (Left) A graph with a regular structure and that contains about 20 000 elements (vertices and edges). (Right) A complex network with about 1 200 000 elements.

The architecture of BrowVis consists of the graph processing pipeline described below. At high-level, once the input graph is loaded, the first step is the computation of a layout, followed by an initial rendering of the computed layout. Subsequent interactions with the user interface may trigger further repetitions of the rendering step.

3.1. Layout

As discussed in Section 2, there exists a vast literature concerning force-directed algorithms, and the design of an original layout algorithm is beyond the scope of this paper. Instead, our choice is to rely on the OGDF implementation of FM^3 [20, 21], a robust implementation already experimented in multiple works. We ported the C++ implementation of FM^3 into WebAssembly, an open standard that defines a portable binary-code format for executable programs and that provides a JavaScript API. In particular, we used Emscripten, an open source software to compile C and C++ code into WebAssembly; the output code is compact and runs at near-native speed. To avoid blocking the user interface during the layout computation, we use a dedicated Web worker that runs in background. As it will be further clarified in the remainder of this section, the layout step of the pipeline is executed only once in BrowVis, whereas the rendering step may be repeated multiple times depending on user interaction.

3.2. Rendering

To perform the rendering, BrowVis engineers the main ideas behind LaGO [12]. The original implementation of LaGO exploits the OpenGL rendering pipeline, which is not conceived for Web browsers. Instead, BrowVis exploits the WebGL technology, which is based on OpenGL ES, a subset of OpenGL. Specifically, we utilize pixi.js, a general-purpose library that offers low-level primitives for 2D rendering. Moreover, the computed layout is rendered by means of a dedicated Web worker that runs in background.

At high-level, BrowVis first accumulates vertices based on density fields and it then exploits the obtained fields to aggregate edges. To accumulate vertices, the system adopts a kernel density estimation (KDE) with Gaussian kernels (hence following the approach in [12]). Formally, let $G = (V, E)$ be a graph with n vertices and m edges and let Γ be a drawing of G in output from

Figure 2: Color palette used for the density field.

the layout step. For each vertex $v_i \in V\,(i \in \{1, \ldots, n\})$, let $p_i = (x_i, y_i)$ be the point representing v_i in Γ. For each pixel $p = (x, y)$ of our drawing area, the density field function at p is defined as follows and depends on the parameter σ:

$$D_f(x, y, \sigma) = \sum_{i=1}^{n} \frac{1}{2\pi\sigma^2} e^{-\frac{1}{2\sigma^2}(x-x_i+y-y_i)^2}$$

Based on the zoom level, part of the drawing Γ may be outside the drawing area; in such a case, the density field is computed only with respect to the vertices that are part of the visible area, plus those vertices that lie in a frame of fixed size around it. Once a density value has been computed for each pixel, the values are normalized in the range $[0, 1]$ and discretizes by using a constant number of levels mapped to the color palette illustrated in Fig. 2. To speed-up our implementation, rather than applying the above formula for each pixel and for each vertex, we generate a single density field prototype and move it iteratively on top of each vertex in the drawing area, in order to update only the pixels in a neighborhood of that vertex.

To aggregate edges, the idea is to identify clusters in the density field and only represent inter-cluster edges. We use a hill climbing algorithm to move each endpoint of an edge to the highest point around it, called *peak* in the following. After this operation, there will be bundles of aggregated edges (those whose endpoints are mapped to the same peaks). In particular, inner-cluster edges (those whose endpoints are both mapped to the same peak) disappear, while inter-cluster edges are emphasized. Let the *weight* of an inter-cluster edge be the number of original edges aggregated into this edge. Similarly as for the density field, the weights are normalized and discretized by using a constant number of levels mapped to the color opacity and to the line thickness associated with the edge segment (whose color is orange). In terms of implementation, for each vertex v_i we identify its cluster by applying the following process. Initially, let p_i be the pixel representing the position of v_i, and consider the 8-neighborhood of p_i. Let q_i be the pixel of this 8-neighborhood with highest density field value. If the value of p_i is larger than q_i, then p_i will be the center of the cluster and the process halts. Otherwise, we set $p_i = q_i$ and we repeat the process.

3.3. Interaction and User Interface

The system provides a user interface with general-purpose interaction features[2]. Once the input graph is loaded, the produced visualization is scaled so to fit entirely within a canvas of fixed size. The user interface makes it possible to obtain coarser or finer vertex and edge aggregations, by suitably modifying the aggregation parameters of the density field and of the hill climbing algorithm. Fig. 3 shows two visualizations obtained by modifying the vertex aggregation level.

[2]A demo version with a preloaded network is available at: http://mozart.diei.unipg.it/montecchiani/browvis/

Figure 3: The same network layout with two different vertex aggregation levels. The network has about 25 000 elements, and the vertex aggregation level has been doubled in the right figure.

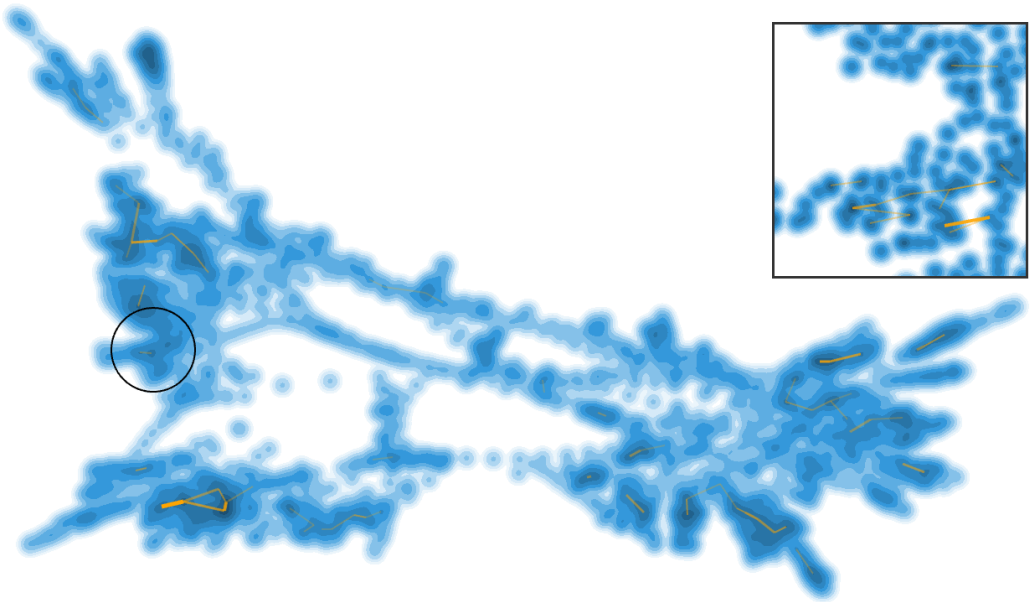

Figure 4: Zooming in a specified portion of the drawing.

A classic zoom and pan feature permits to move the drawing (panning) with respect to the canvas, or to scale it up and down (zooming). As a consequence of a zoom or pan operation, the sets of vertices and edges in the canvas change and both the density field and the edge aggregation are recomputed. In other words, the rendering step is repeated on a portion of the whole layout. When zooming-in, the density field becomes more fine grained and fewer edges are aggregated, while when zooming-out, the density field becomes less detailed and more edges are bundled together. An undesired effect of a quick zoom-in or zoom-out operation is a sudden change in the visualization, which is caused by the sharp (dis)aggregation of vertices and edges. To cope with this issue, an important feature of our interface, is that the vertex and edge aggregation levels are dynamically tuned so to make the whole exploration process more stable. We remark that this feature is not present in [12].

Table 1
Details for the `Real` benchmark. $|V|$ and $|E|$ are the number of vertices and edges of the instances.

Name	$\lvert V\rvert$	$\lvert E\rvert \downarrow$	Description
add32	4 960	9 462	circuit simulation problem
ca-GrQc	5 242	14 496	collaboration network
poli_large	15 575	17 427	chemical process simulation
p2p-Gnutella04	10 876	39 994	P2P network
pGp-giantcomp	10 680	48 632	Pretty-Good-Privacy network
ca-CondMat	23 133	93 497	collaboration network
p2p-Gnutella31	62 586	147 892	P2P network
ASIC_320ks	321 523	515 300	circuit simulation problem
amazon0302	262 111	899 792	co-purchasing network
com-amazon	334 863	925 872	co-purchasing network
com-DBLP	317 080	1 049 866	collaboration network
roadNet-PA	1 087 562	1 541 514	road network

A second feature allows the user to select a smaller portion of the drawing, which is rendered inside a dedicated view with a zoom level chosen by the user and independent of the zoom level of the whole drawing; see Fig. 4 for an illustration.

Finally, BrowVis makes it possible to display a certain percentage of the node labels with higher degree; this percentage is automatically set by the system based on the current zoom level and on the current level of vertex aggregation. In particular, to limit the overall visual complexity, one can choose to visualize the labels in a neighborhood of a desired point.

4. Experimental Analysis

Graph benchmark. We used two different benchmarks of graphs, already exploited in similar experiments (see, e.g., [10]).

–`Real`. It consists of 12 real networks, with up to 1.5 million edges, taken from the Sparse Matrix Collection of the University of Florida[3], the Stanford Large Networks Dataset Collection[4], and the Network Data Repository[5] [44]. Details about name, type, and structure of these graphs are reported in Table 1. The whole algorithmic pipeline (layout and rendering) is applied on these graphs after the removal of isolated vertices, self-loops, and parallel edges.

– `Synth-Rand`. It contains 18 synthetic random graphs generated with the Erdõs-Rényi model [45]. These graphs are divided into six groups of three graphs each, with size (number of edges) $m \in \{10^4, 5 \cdot 10^4, 10^5, 10^6, 1.5 \cdot 10^6, 2 \cdot 10^6\}$ and density (number of edges divided by

[3] http://www.cise.ufl.edu/research/sparse/matrices/
[4] http://snap.stanford.edu/data/index.html
[5] http://www.networkrepository.com/

Table 2
Running time and memory footprint. The symbol σ denotes the standard deviation.

		LAYOUT		RENDERING		TOTAL			
		Time [ms]		Time [ms]		Time [ms]		Memory [MB]	
	Graph Name	Mean	σ	Mean	σ	Mean	σ	Mean	σ
Real	add32	2 406	45	185	2	2 591	47	17	0.1
	ca-GrQc	2 440	41	233	4	2 673	44	17	0.2
	poli_large	6 942	69	326	15	7 268	78	41	0.3
	p2p-Gnutella04	6 473	26	312	6	6 785	23	30	0.1
	pGp-giantcomp	6 788	79	303	15	7 091	93	30	0
	ca-CondMat	12 798	38	435	29	13 233	66	60	0.3
	p2p-Gnutella31	40 750	397	956	7	41 707	397	150	0
	ASIC_320ks	210 625	518	3 747	59	214 373	467	747	0.5
	amazon0302	160 928	849	3 620	33	164 548	860	620	0.5
	com-Amazon	220 228	298	4 505	49	224 793	285	788	0.5
	com-DBLP	214 869	1 146	4 479	78	219 349	1 219	753	0.5
	roadNet-PA	778 083	18 037	14 110	123	792 193	17 957	2 631	79
	# Edges								
Synth-Rand	10 000	2 163	137	219	9	2 383	145	14	1
	50 000	14 229	2 222	479	25	14 708	2 243	57	4
	100 000	26 735	7 686	720	82	27 455	7 768	99	14
	1 000 000	371 526	50 525	6 469	129	377 996	50 646	1 116	12
	1 500 000	501 533	66 731	8 850	826	510 383	67 556	1 507	153
	2 000 000	644 480	40 911	11 653	464	656 133	41 358	2 041	192

number of vertices) in the range $[2, 3]$.

Experimental setting. We executed the experiments on a MacBook Pro (Mid 2015) laptop equipped with an i7-4870HQ CPU, 16 GB of RAM, and running macOS Big Sur as operating system. Also, for the experiments we used the 96.0.4664.45 version of the 64-bit Google Chrome browser. For each computation, we measured the running time and the memory footprint. For each graph, we repeated the computation three times.

Results. Table 2 shows the recorded running time and memory footprint for each of the two benchmarks. The running time is split between the two main steps, layout and rendering. The values are averaged over the three executions. For `Synth-Rand`, the graphs are further grouped based on the number of edges. The standard deviation is also reported.
– `Real`. One can observe that the layout step is about one order of magnitude slower than the rendering step. The relatively small standard deviation values assess a good stability of the algorithms. The smallest network ($\approx 15 \cdot 10^3$ elements) took about 2.5 seconds to be visualized, while the largest one ($\approx 2.5 \cdot 10^6$ elements) took about 13 minutes. Notably, the rendering step took less than 1 second for all graphs with up to about $2 \cdot 10^5$ elements, and about 14 seconds for

the largest instance. Concerning the primary memory required by the computations, it ranges from 17 MB for the smallest graph to about 2.6 GB for the largest instance.
– `Synth-Rand`. Again the layout is significantly slower than the rendering. The standard deviation is large, due to the fact that graphs in the same group can have (slightly) different sizes. However, the more uniform structure of the graphs yields faster runtimes. The smallest instances ($\approx 14 \cdot 10^3$ elements) took 2.2 seconds to be visualized, those with $\approx 2 \cdot 10^6$ elements took about 8 minutes, while the largest ones ($\approx 2.8 \cdot 10^6$ elements) took about 11 minutes.

Discussion. Our experiments show that BrowVis is able to visualize graphs with several thousand edges in seconds, while it can scale up to graphs with millions of edges in minutes. We remark that, once the initial visualization has been computed, any further interaction only requires to (partially) repeat the rendering step, which never took more than 15 seconds in our experiments, and it actually took less than 0.5 seconds for all instances with less than 10^5 edges. In terms of scalability, it shall be noticed that, since the WebAssembly code runs in a sandbox, the communication with JavaScript is obtained via shared memory locations references with 32-bit pointers, which limits the maximum amount of primary memory that can be used to 4GB. Hence, scaling to much larger graphs would require to overcome this memory limit, e.g., by sparsifing or filtering the graph in a preprocessing routine [46]. As already discussed, the experiments conducted in [19] report neither the running time nor the memory footprint of the algorithms. In particular, it is not clear what it is the initial time to wait until a first stable visualization is produced. Yet, the experiments in [19] suggest that BrowVis outperforms the considered technologies, namely Stardust.js and D3.js can render up to a total of 10^5 elements (both vertices and edges), while NetV.js can render up to a total of 10^6 elements showing at least 1 frame per second.In addition, we remark that BrowVis produces an interactive abstraction of the input graph, which allows for a details-on-demand exploration.

5. Future Work

Our work demonstrates that visualization pipelines of considerably large graphs can be entirely integrated in client-side systems. There are several research directions that are worth pursuing:

- The most natural research direction is to further speed-up our techniques. We believe that pursuing such a direction, especially for the rendering step, requires the design of more sophisticated data structures to update the density field and the edge bundles dynamically. In addition, alternative exploration paradigms or visual abstraction methods may contribute to this research direction.
- We focused on static graphs whose structure does not change over time. Dealing with dynamic graphs, such as those originated by streaming data sources, would open the way to new interesting applications (see, e.g., [23]). This would require to re-think both the layout step, which should produce layouts that remain stable over time, as well as the rendering step, which should highlight the changes that occur over time.
- Integrating intelligent agents that employ task offloading strategies [47] is also of great interest, in order to exploit a broader and dynamic range of available computing resources, spanning from local hardware to the cloud through edge devices.
- Finally, we plan to further evaluate the system with domain users and experts.

References

[1] X. Jia, T. Wen, W. Ding, H. Li, W. Li, Semi-supervised label distribution learning via projection graph embedding, Inf. Sci. 581 (2021) 840–855.

[2] W. Didimo, L. Grilli, G. Liotta, F. Montecchiani, D. Pagliuca, Visual querying and analysis of temporal fiscal networks, Inf. Sci. 505 (2019) 406–421.

[3] Z. Lin, L. Feng, R. Yin, C. Xu, C. K. Kwoh, GLIMG: global and local item graphs for top-n recommender systems, Inf. Sci. 580 (2021) 1–14.

[4] N. Binesh, M. Ghatee, Distance-aware optimization model for influential nodes identification in social networks with independent cascade diffusion, Inf. Sci. 581 (2021) 88–105.

[5] R. Baggio, M. Fuchs, Network science and e-tourism, J. Inf. Technol. Tour. 20 (2018) 97–102.

[6] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, M. T. Özsu, The ubiquity of large graphs and surprising challenges of graph processing: extended survey, VLDB J. 29 (2020) 595–618.

[7] S. Cheong, Y. Si, Force-directed algorithms for schematic drawings and placement: A survey, Inf. Vis. 19 (2020).

[8] Y. Hu, L. Shi, Visualizing large graphs, Wiley Interdisciplinary Reviews: Computational Statistics 7 (2015) 115–136.

[9] S. G. Kobourov, Force-directed drawing algorithms, in: R. Tamassia (Ed.), Handbook of Graph Drawing and Visualization, CRC, 2013.

[10] A. Arleo, W. Didimo, G. Liotta, F. Montecchiani, Large graph visualizations using a distributed computing platform, Inf. Sci. 381 (2017) 124–141. doi:10.1016/j.ins.2016.11.012.

[11] A. Arleo, W. Didimo, G. Liotta, F. Montecchiani, A distributed multilevel force-directed algorithm, IEEE Trans. Parallel Distributed Syst. 30 (2019) 754–765.

[12] M. Zinsmaier, U. Brandes, O. Deussen, H. Strobelt, Interactive level-of-detail rendering of large graphs, IEEE Trans. Vis. Comput. Graph. 18 (2012) 2486–2495.

[13] K. El Gebaly, J. Lin, In-browser interactive SQL analytics with afterburner, in: SIGMOD Conference, ACM, 2017, pp. 1623–1626.

[14] J. Lin, Building a self-contained search engine in the browser, in: ICTIR, ACM, 2015, pp. 309–312.

[15] J. Lee, R. Tang, J. Lin, Honkling: In-browser personalization for ubiquitous keyword spotting, in: EMNLP/IJCNLP (3), Association for Computational Linguistics, 2019, pp. 91–96.

[16] B. Breve, L. Caruccio, S. Cirillo, V. Deufemia, G. Polese, Dependency visualization in data stream profiling, Big Data Res. 25 (2021) 100240.

[17] L. Caruccio, S. Cirillo, V. Deufemia, G. Polese, Real-time visualization of profiling metadata upon data insertions, in: C. Costa, E. Pitoura (Eds.), EDBT/ICDT 2021, volume 2841 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2021.

[18] F. J. Villanueva, C. Aguirre, A. Rubio, D. Villa, M. J. Santofimia, J. C. López, Data stream visualization framework for smart cities, Soft Comp. 20 (2016) 1671–1681.

[19] D. Han, J. Pan, X. Zhao, W. Chen, Netv.js: A web-based library for high-efficiency visualization of large-scale graphs and networks, Vis. Informatics 5 (2021) 61–66.

[20] M. Chimani, C. Gutwenger, M. Jünger, G. W. Klau, K. Klein, P. Mutzel, The open graph drawing framework (OGDF), in: Handbook of Graph Drawing and Visualization, Chapman and Hall/CRC, 2013, pp. 543–569.

[21] S. Hachul, M. Jünger, Large-graph layout algorithms at work: An experimental study, J. Graph Algorithms Appl. 11 (2007) 345–369.

[22] T. von Landesberger, A. Kuijper, T. Schreck, J. Kohlhammer, J. J. van Wijk, J. Fekete, D. W. Fellner, Visual analysis of large graphs: State-of-the-art and future research challenges, Comput. Graph. Forum 30 (2011) 1719–1749.

[23] S. Cheong, Y. Si, R. K. Wong, Online force-directed algorithms for visualization of dynamic graphs, Inf. Sci. 556 (2021) 223–255.

[24] W. Didimo, F. Montecchiani, Fast layout computation of clustered networks: Algorithmic advances and experimental analysis, Inf. Sci. 260 (2014) 185–199.
[25] U. Dogrusöz, E. Giral, A. Cetintas, A. Civril, E. Demir, A layout algorithm for undirected compound graphs, Inf. Sci. 179 (2009) 980–994.
[26] T. M. J. Fruchterman, E. M. Reingold, Graph drawing by force-directed placement, Softw. Pract. Exp. 21 (1991).
[27] A. J. Quigley, P. Eades, FADE: graph drawing, clustering, and visual abstraction, in: GD 2000, volume 1984 of *LNCS*, Springer, 2000, pp. 197–210. doi:10.1007/3-540-44541-2_19.
[28] R. Hadany, D. Harel, A multi-scale algorithm for drawing graphs nicely, Discrete Appl. Math. 113 (2001) 3–21.
[29] C. Walshaw, A multilevel algorithm for force-directed graph-drawing, J. Graph Algorithms Appl. 7 (2003) 253–285.
[30] S. Hachul, M. Jünger, Drawing large graphs with a potential-field-based multilevel algorithm, in: GD 2004, volume 3383 of *LNCS*, Springer, 2004, pp. 285–295.
[31] G. Bartel, C. Gutwenger, K. Klein, P. Mutzel, An experimental evaluation of multilevel layout methods, in: GD 2010, volume 6502 of *LNCS*, Springer, 2010, pp. 80–91.
[32] S. Ingram, T. Munzner, M. Olano, Glimmer: Multilevel MDS on the GPU, IEEE Trans. Vis. Comput. Graph. 15 (2009) 249–261.
[33] H. Meyerhenke, M. Nollenburg, C. Schulz, Drawing large graphs by multilevel maxent-stress optimization, IEEE Trans. Vis. Comput. Graph. PP (2017) 1–1. doi:10.1109/TVCG.2017.2689016.
[34] A. Hinge, G. Richer, D. Auber, MuGDAD: Multilevel graph drawing algorithm in a distributed architecture , in: WSCG 2017, 2017, p. 189.
[35] B. Shneiderman, The eyes have it: A task by data type taxonomy for information visualizations, in: VL, IEEE Computer Society, 1996, pp. 336–343.
[36] J. Abello, F. van Ham, N. Krishnan, Ask-graphview: A large scale graph visualization system, IEEE Trans. Vis. Comput. Graph. 12 (2006) 669–676.
[37] D. Auber, Y. Chiricota, F. Jourdan, G. Melançon, Multiscale visualization of small world networks, in: INFOVIS, IEEE Computer Society, 2003.
[38] F. v. Ham, J. J. van Wijk, Interactive visualization of small world graphs, in: INFOVIS, IEEE Computer Society, 2004, pp. 199–206.
[39] A. Perrot, D. Auber, Cornac: Tackling huge graph visualization with big data infrastructure, IEEE Trans. Big Data 6 (2020) 80–92.
[40] 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 (2016) 309–311.
[41] M. Bostock, V. Ogievetsky, J. Heer, D^3 data-driven documents, IEEE Trans. Vis. Comput. Graph. 17 (2011) 2301–2309.
[42] J.-P. Coene, sigmajs: An R htmlwidget interface to the sigma.js visualization library, . Open Source Softw. 3 (2018) 814.
[43] D. Ren, B. Lee, T. Höllerer, Stardust: Accessible and transparent GPU support for information visualization rendering, Comput. Graph. Forum 36 (2017) 179–188.
[44] R. A. Rossi, N. K. Ahmed, The network data repository with interactive graph analytics and visualization, in: AAAI 2015, 2015. URL: http://networkrepository.com.
[45] P. Erdõs, A. Rényi, On random graphs I, Publicationes Mathematicae 6 (1959) 290–297.
[46] X. Huang, C. Huang, NGD: filtering graphs for visual analysis, IEEE Trans. Big Data 4 (2018) 381–395.
[47] F. Saeik, M. Avgeris, D. Spatharakis, N. Santi, D. Dechouniotis, J. Violos, A. Leivadeas, N. Athanasopoulos, N. Mitton, S. Papavassiliou, Task offloading in edge and cloud computing: A survey on mathematical, artificial intelligence and control theory solutions, Comput. Networks 195 (2021) 108177.