A Lightweight Tool to Visualize Dynamic Networks

Mar Jovani Albalat, Valerio Tonelli, Nicoleta Cuprinsu, Dawid Weglarz, Max Roeters, Michael Burch
Eindhoven University of Technology

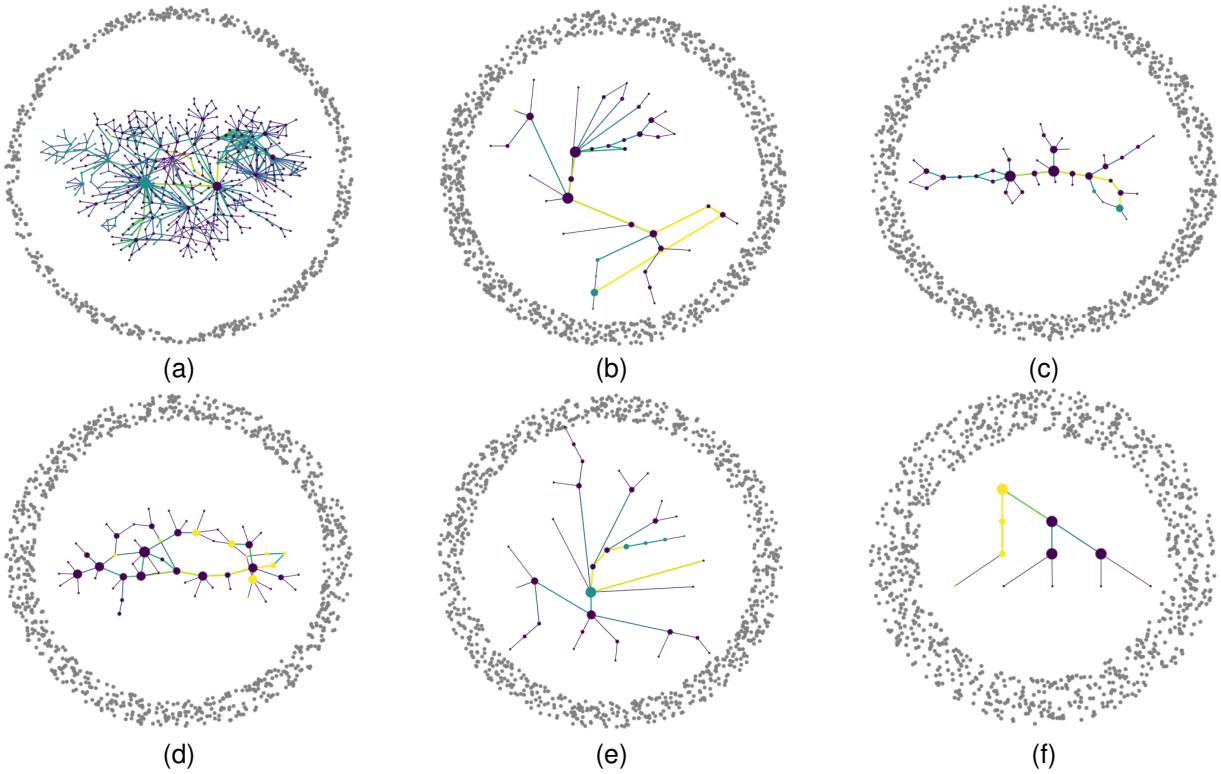

Figure 1: The node-link graph for the same dynamic graph at different time steps (respectively 1, 514, 430, 425, 536, 293) using force-directed, hierarchical, and radial layouts. Notice the difference in number of isolated nodes.

ABSTRACT

Visualization has many applications in the analysis of networks, where it provides an intuitive view of vast amounts of information at a glance. As datasets increase in complexity and evolve faster, the need to study both their static patterns and their dynamics over time is becoming ever more necessary. However, while there are plenty of tools and disparate techniques available to the former end, the same cannot be said for the latter. In this paper we present such a tool, which is web-based, easy to use, simple and lightweight, for small to medium-sized dynamic graphs. We clarify its usage, illustrate its *modus operandi* and demonstrate its usefulness on a dataset of approximately 30,000 edges taken from the trace-back of a program execution.

Index Terms: Human-centered computing—Visualization—Visualization design and evaluation methods

1 INTRODUCTION

When people tend to describe phenomena in the outside world, they often simplify them. That is, they reduce or ignore some of their complexity, in order to make computations faster, and reasoning and communication easier. Among these simplifications, reduction to a static network is a suitable technique.

Commonly, when we think about 'networks', we immediately picture friendships on social media, navigation paths, or protein structures. However, while one can infer many properties from a snapshot of these systems at some point in time, one can also notice that friendships frequently change, that new paths (streets, highways) are often built, and that molecules modify their bonds under certain conditions [2]. These are variations which lead us beyond the static representation of data: what if it is just these dynamics between those static situations which yield the most information? In these instances, we need to introduce one dimension of analysis from our simplification back–time.

Dynamic networks are a young field of study, which can open new opportunities in the field of data science when it comes to analyzing and identifying change within and across networks. Given the ever-growing complexity of even the most trivial of modern systems [24], they can provide a unique insight in various illustrative problems: social media analysis [25], assessment of public health systems [22], or even of terror groups [21].

In this paper, such a tool is presented. The application renders data, formatted as a dynamic network, into several visualizations, and provides an interactive environment that the user can adjust to his/her wishes. More details regarding the data format, the implementation, and how to execute the application are in Sections 3 and 4.

2 Previous Work

Graphs constitute a powerful conceptual tool, both in the sense of mathematical abstraction and of their visual counterpart, as they are an intuitive and versatile model to express relationships between data. Consequently, *Graph Theory* is a rich and developed field of study, which finds innumerable applications, such as semantic networks in linguistics, social networks analysis, or species and habitat modelling in biology. As Beck et al. [3] mention, researchers have often addressed the formal and visual limitations of graphs, the latter with respect to cluttering and, notably, time-dependent datasets.

Furthermore, many visualization techniques have also been explored. On this topic, Eades et al. [17] stress that rearrangements on visualizations must not disturb the so-called *mental map* that the user has formed of the data representation, which is deemed critical in maintaining its intrinsic value. Misue et al. [23] have built upon this idea, discussing layout adjustment methods for graphs. Another similar study, but wider in scope, with no restrictions on the class of graphs or the type of layout algorithm, is by Friedrich and Eades [19].

After the millennium, techniques specially designed to visualize dynamic graphs began to emerge, due to the presence of more and larger time-dependent datasets. Two approaches to this visualization can be distinguished: **animation**, which is self-explanatory, and **time-to-space mappings** that produce a static image. Comparing those two methods, with regards to how well they create and keep a mental map for the viewer, spawned a new field of research, as Archambault et al. [1] discuss. Another study which looked at the advantages and disadvantages of animation and static representation of dynamic graphs was by Farrugia and Quigley [18], which found that static representations are generally more effective.

To further build on static representations for dynamic graphs, researchers tried to entail more and more information without being detrimental to the mental map [7–9, 12, 13, 26]. While a paper by Branke [5] introduces and explains key points that should be considered in this field, and suggests new avenues for further research, Diehl and Görg [16] present a generic algorithm for drawing sequences of graphs and introduce a new way of computing graph layouts.

A quite interesting visualization technique is described by Burch et al. [11]. Their approach has the benefit of showing a concurrent overview of vertices, edges, and time steps all in a single image thanks to the usage of **bipartite graphs**. An even more advanced technique is discussed in Bruder et al. [6], which presents an approach to interactively analyze large dynamic graphs, with up to several thousands of time steps.

Overall, there has been considerable research on how to visualize dynamic graph data in such a way that it balances presenting as much data as possible while also being understandable and, where possible, precognitive. Some researchers focus on how to transform dynamics into a static picture, some work on the representation itself, finally others handle the interpretation of that representation. Classifications or summaries of all these techniques are presented by the works of Casteigts et al. [14] and Beck et al. [3].

The visualization tool presented in this paper combines many of the mentioned techniques and attempts to bundle them in an easy-to-work and clear environment. For the most part, it builds on the work of some classic visualization techniques [10], such as node-link diagrams and adjacency matrices, as well as the aforementioned concept for scalable time-to-space mapping for dynamic datasets by Burch et al. [11], which is discussed more in-depth in Section 4.2.1. Furthermore, it makes use of several known algorithms and commonly used Python libraries, as well as a novel idea for the layout of isolated nodes, which produces graphs akin to those in Figure 1. All of these are further discussed in Sections 4.3 and 7.

3 Data Model and Data Preprocessing

In this section we will focus on describing the meaning and purpose of dynamic network data, the format which encodes this type of data, and how our tool supports it.

3.1 Data Model

Formally, we define the following:

- **Directed graph:** A triple $G = (V, E, w)$, where V is a set of vertices, E is any subset of the Cartesian product $V \times V$, and $w : E \to \mathbf{R}$ is the weight function.
- **Dynamic graph:** An array of directed graphs, that is, a vector D_t, where t is the number of time steps.
- **Adjacency matrix:** A matrix $M_{n,n}$ where, given a graph $G = (V, E, w)$, n is the number of vertices of G, and element m_{ij} indicates whether node pair (i, j) belongs to E. Typically, value 0 is used to indicate a lack of connection, while any other real value x indicates a direct link with weight $w(i, j) = x$. From the definition it is also clear that rows depict starting nodes and columns depict target nodes. If the dataset is non-directed, the matrix is symmetric, otherwise the matrix will be asymmetric.

3.2 Input Parsing

A line-based text file offers a convenient solution to store data in order to construct a dynamic, directed, and weighted graph. We require a text file as input, with its lines having the following format:

timestamp	start	end	weight
int	**int**	**int**	**float**
...	...	...	...

It should be noted that surmising patterns from this plain format, even for small to medium sized datasets, is painstakingly slow, if not outright impossible, due to the limits in the cognitive abilities of the human brain to remember and comprehend a large amount of text. In comparison, a visualization is a much more efficient way to infer any interesting property out of these datasets [15].

An additional text file can be supplied to add a hierarchy to each node, where the line number corresponds to the node number:

```
hierarchy_for_node_1
hierarchy_for_node_2
...
```

Once the tool receives this input, it creates an array of empty *networkx* graphs, one for each time step, and fills in the edges by reading the input file one line at a time, also creating new vertices when needed. If a metadata file has also been supplied, the hierarchy is stored on the vertices as an additional attribute.

4 The Application

This section focuses on how to run our application, how to navigate its interface, which visualizations and filters are offered, how to interpret them, and what can be deduced from them.

Let us discuss the design philosophy behind our tool first. We have built it in such a way to ensure that it is:

- **Compact:** All visualizations can be seen and explored by the user within a single screen.
- **Essential:** Frills and visual candy is reduced to a minimum outside of the visualizations, to ensure a proper user focus and readability.
- **Hierarchical:** There is a context-to-focus hierarchy between the visualizations.

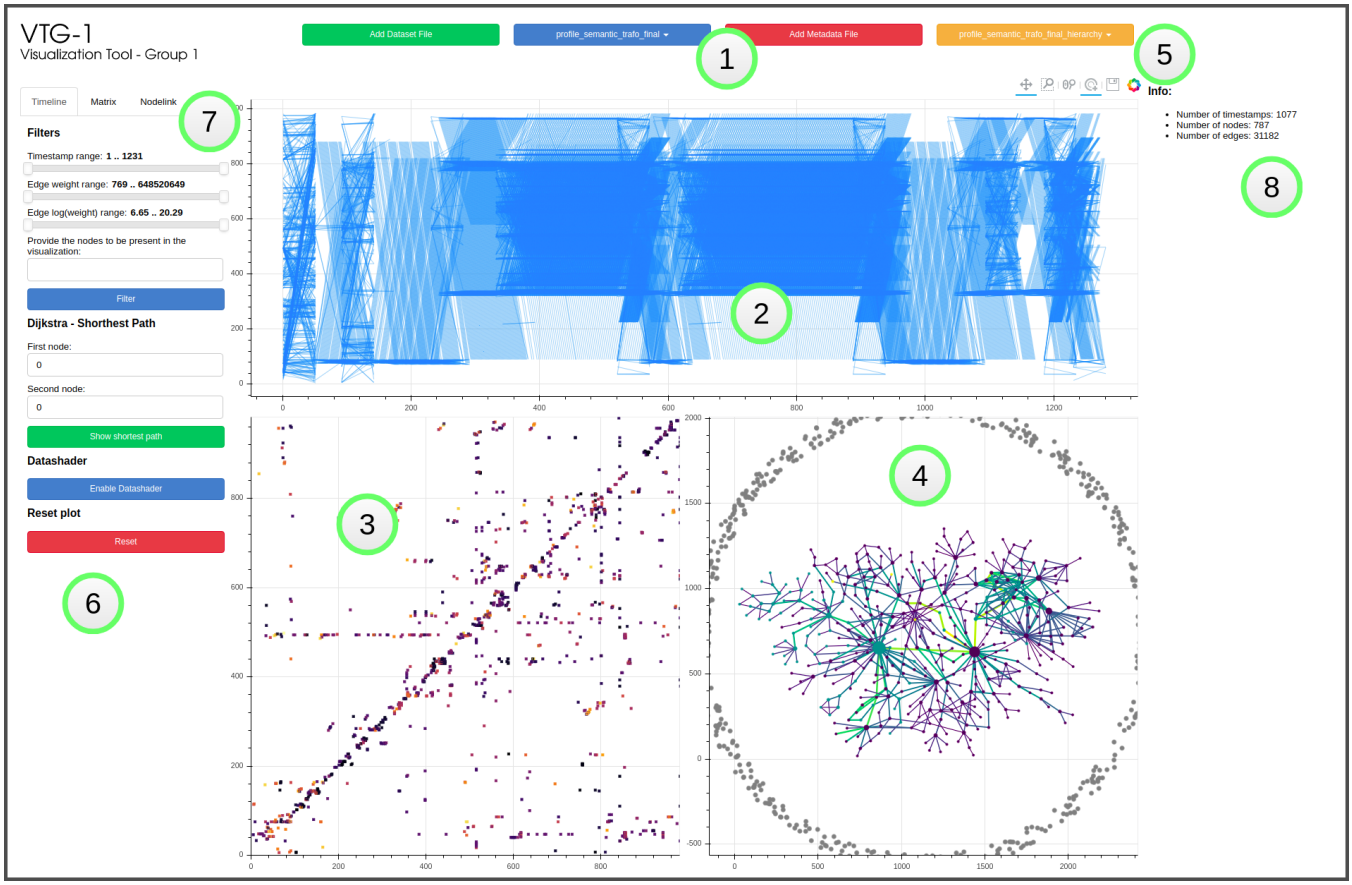

Figure 2: A screenshot of the application, with numbers indicating its elements.

- **Interactive:** The visualizations allow user selection and filtering of their elements.
- **Fast:** Time complexity is minimized for all algorithms within reasonable limits, trading off memory when necessary.

4.1 Graphical User Interface

The program runs as a *bokeh* server. From the folder containing the Python files, the user can run the following command to start the application:

```
$ bokeh serve --show .
```

For more information regarding necessary libraries and packages and how to install them, the user can refer to the included *README.md* file.

With reference to Figure 2, the user can open a dataset by pressing the *Load Dataset* button at the top of the application (1), or select one from the drop-down list. Any newly loaded dataset is copied onto the server, such that it becomes quickly available the next time. The metadata file can be loaded in much the same manner.

The central area of the application is reserved for the three visualizations (2), (3), and (4). At the top of this area, in the top right corner, the *bokeh* toolbar (5) is present, which allows the user to switch between pan, zoom on the visualizations, and other operations such as resetting the view or saving them to an image file.

The left side of the application (6) is reserved for inputs that modify the visualizations, such as edge/vertices filtering, choosing a preferred palette, or applying an algorithm like the shortest path. The affected visualization can be changed by using the tabs at the top (7), and as the user switches between them so do the options in this area.

Finally, the right hand side (8) is dedicated to a simple text-based output, indicating the status of the application regarding the loading of datasets and the execution of algorithms.

4.2 Visualization Techniques

Currently, the application provides three different plots: a time-context visualization in the form of a **dynamic timeline**, a clustered and aggregated visualization through an **adjacency matrix**, and finally, a typical **node-link diagram** for a pinpoint, discrete vertices/edges analysis.

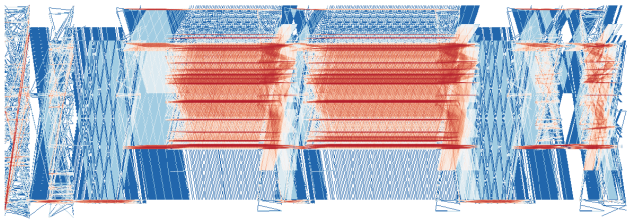

Figure 3: The dynamic timeline with *datashader* enabled.

4.2.1 Dynamic Timeline

The dynamic timeline constitutes the main view of the application. It is the starting point and the greatest context for all further analysis,

and it is implemented through a series of interleaved graphs in a 2D space, as described by Burch et al. [11] and discussed more in depth in Section 4.3.

Upon this image, by itself static, we also allow some degree of user interactivity: filtering by edges, nodes, edge weights, or any combination of the three, zoom in at an arbitrary level of detail and select edges of interest. The hovering interaction provides additional information about all the edges available in the visualization, without overlapping content when the edges are cluttered.

This visualization approach proves to be general enough to render an effective picture for most graphs, while being trivial to code and extendable to any reasonable amount of time steps. Moreover, since it foregoes techniques such as animation or 3D mapping, both of which would carry heavy tradeoffs [3], it provides a precognitive, intuitive understanding of the dynamic patterns hidden within the input dataset.

In regards to its interactivity, it should be noted that the zoom-ins do not always look appealing and may not provide a clearer view of the dynamics of the graph, since the amount of link crossings may, in all likelihood, create a cluttered image.

To lessen this effect, a static *datashader* render can be produced for the current view of the timeline, so that the color of each pixel is on a gradient based on the density of link crossings at that point. An example of such rendering can be seen in Figure 3.

4.2.2 Adjacency Matrix

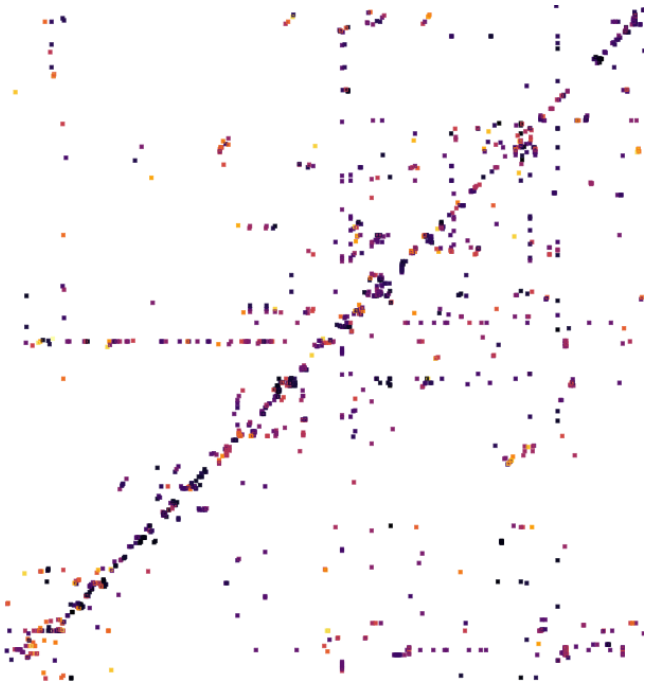

Figure 4: The aggregated adjacency matrix.

The adjacency matrix is visible in the second view of the application, and it is an aggregated matrix over a range of time steps defined by the user. The aggregate function is the logarithm of the sum of the weights, and these numeric values are then mapped to a color palette.

One of the advantages of this kind of visualization is that it does not suffer from occlusion and link crossings as node-link diagrams do [4]. The tradeoff is in higher-level tasks, such as identifying groups or highly-connected vertices, as these operations do require a reordering of rows (and columns) to reveal patterns. Algorithmic solutions to this, as well as the aggregation techniques, are discussed in Section 4.3.4. A rendering of the matrix can be seen in Figure 4.

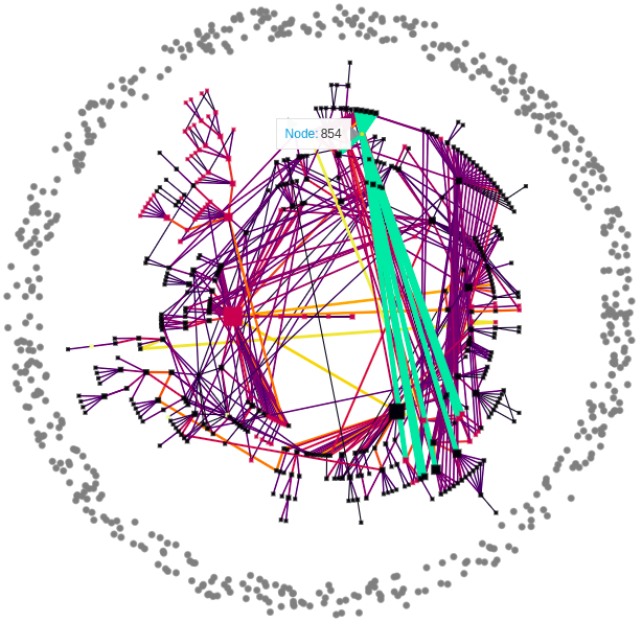

Figure 5: *twopi* layout of the node-link diagram, with square nodes and *Inferno* color palette. As a result of hovering over node 854, all edges connected to it are highlighted.

4.2.3 Node-Link Diagram

A classic node-link graph is available in the third view. By default, the application uses a force-directed layout optimized for large graphs called **sfdp**. However, the user can change the layout to a hierarchical layout named **dot**, or to a radial layout named **twopi**; all three are provided by the *Graphviz* library [20].

Whichever vertex positioning method is chosen, it is hybridized with a circular layout, such that the isolated nodes surround the connected graph(s). A detailed description of this placement algorithm is discussed in Section 4.3.5.

We have implemented typical visualization techniques for this graph, such as node coloring and node size proportional to the number of connections, edge coloring and node thickness based on the edge weight, several color palettes, and color highlighting when used in tandem with algorithms. Furthermore, the circular layout allows the user to estimate the ratio between connected and disconnected nodes, an important property for sparse graphs.

A few resulting renders of this visualization can be seen in Figure 1, while an application of the hovering functionality can be seen in Figure 5.

4.3 Algorithms

In this section we focus on introducing the main issues that arise while analyzing dynamic networks, offering clear algorithmic solutions.

4.3.1 Dynamic Timeline

The algorithm behind the dynamic timeline renders a dynamic graph onto a single 2D image by projecting each static graph in a 1D space. In particular, the visualization encodes single time steps on a graph where all vertices are displayed in both of two parallel columns, with some constant distance between them (50 pixels) and edges directed from the nodes of the first column to those of the second one. This layout, called a **bipartite graph**, maintains the vertex ordering on

the left-to-right transition and allows easy scalability to an arbitrary amount of vertices and edges. Therefore, it is possible to obtain a meaningful picture of subsequent time steps by interleaving several bipartite graphs, which are superimposed on top of one another except for a single pixel of offset.

4.3.2 Shortest Path Problem

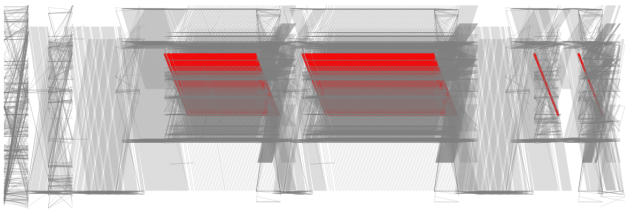

Figure 6: Shortest path (Dijkstra) solution from node 750 to node 450, applied to the dynamic timeline.

The shortest path between any two given nodes is a well-known problem in *graph theory*. Our application makes use of two algorithms that provide efficient solutions to this problem: Dijkstra and Floyd-Warshall, the former being implemented through a Python dictionary and the latter coming from the *networkx* library and being applied as part of the matrix reordering algorithm discussed in Section 4.3.4.

An interesting application of the shortest path problem can be seen in Figure 6: the shortest path between two nodes is calculated for all time steps and thus it is possible to see its general evolution over time as a series of red lines. The node-link graph can also display the shortest path at a given time step.

4.3.3 Graph Aggregation

The adjacency matrix is aggregated over all time steps. That is, given a starting time step $i > 0$ and an end time step $j \leq n$, where n is the maximum time step, the algorithm takes as input an array of *networkx* graphs and produces a single static graph in output, where each edge between nodes a and b of the resulting static graph is determined as the *aggregate function* of all the edges between the same a and b in the various time steps i to j of the dynamic graph. In our case, the aggregate function is the sum of the weights.

4.3.4 Adjacency Matrix Ordering

In order to produce visually appealing matrices and avoid a noise-like pattern, a reordering of the vertices along the columns/rows is needed. We have opted for a hierarchical clustering method [4], given its simplicity of implementation and generally quick results.

Two similar versions of this algorithm–*real* and *avg*–can be executed from our tool, with the idea that they may grant the user more flexibility. This is especially true since, as discussed in Section 6, this algorithm has the highest time complexity.

Given a number p_k of clusters, both algorithms begin as follows. Initially, each vertex is considered a cluster, for a total of $k = n$ clusters where n is the amount of vertices. The shortest distance between all nodes is then calculated and stored in a dictionary, and the algorithms thus enter their main loop.

The *real* version of the algorithm proceeds by finding the two **nodes** with the lowest mutual distance and merging the clusters to which they belong into a single one, and then removes this distance from the dictionary.

The *avg* version, instead, finds the two **clusters** with the lowest distance by averaging the distances of each pairing between the nodes of the start and end clusters, merges the two clusters but does not modify the dictionary.

In both cases, the total amount of clusters k is decreased by one. This is repeated until the amount of clusters $k \leq p_k$. Finally, a permutation is applied to the vertices such that those which belong to the same cluster are subsequent. Algorithms 1 and 2 provide the pertaining pseudo-codes, while Figure 7 can be compared with Figure 4 to see how the reordering enhances the visualization.

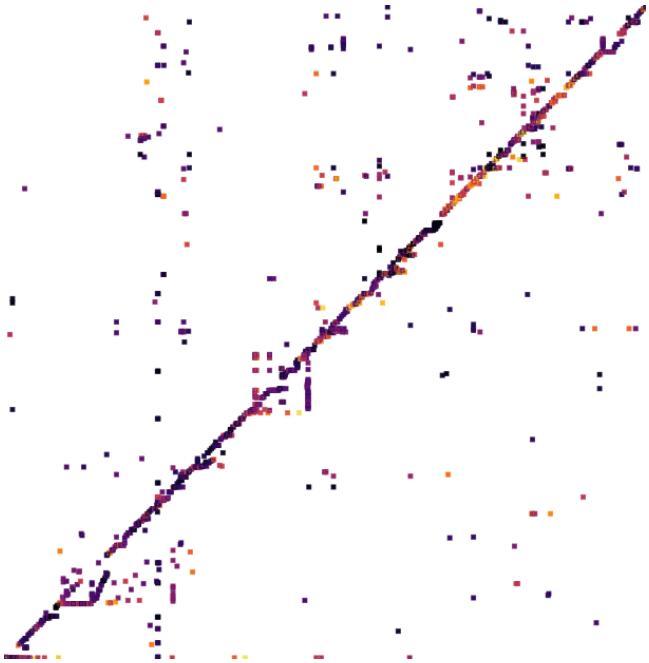

Figure 7: An ordering of the adjacency matrix, using the hierarchical clustering method.

Algorithm 1 Order matrix (*real*)

```
agg_graph ← aggregated graph provided as input
k ← 45 //hard-coded number of final clusters
sp ← shortest path between all pairs of nodes in agg_graph
clusters ← one node per cluster
remove self loops from sp
lc ← number of nodes in agg_graph
while lc > k do
    find minimum distance between all pairs in sp, if it exists
    if min. distance exists then
        min_d ← minimum distance between all pairs in sp
        min_s ← source node from the min. distance pair
        min_t ← target node from the min. distance pair
        cl_s ← cluster to which min_s belongs
        cl_t ← cluster to which min_t belongs
        remove source-target edge from sp
        if cl_t ≠ cl_s then
            copy all nodes from cl_t into cl_s
            lc ← lc - 1
        end if
    else
        break
    end if
end while
iterate over clusters and position nodes incrementally
update plot
```

Algorithm 2 Order matrix (*avg*)

```
agg_graph ← aggregated graph provided as input
k ← 45 //hard-coded number of final clusters
sp ← shortest path between all pairs of nodes in agg_graph
clusters ← one node per cluster
max_distance ← max(sp)
remove self loops from sp
lc ← number of nodes in agg_graph
while lc > k do
    min_d ← +∞
    for cl_1, cl_2 in clusters, with cl_1 ≠ cl_2 do
        sum ← 0
        i ← 0
        for n1 in cl_1, n2 in cl_2 do
            if distance(n1, n2) is finite then
                sum ← sum + distance(n1, n2)
            else
                sum ← sum + max_distance + 1
            end if
            i ← i + 1
        end for
        if sum / i < min_d then
            min_d ← sum / i
            cl_s ← cl_1
            cl_t ← cl_2
        end if
    end for
    copy all nodes from cl_t into cl_s
    lc ← lc - 1
end while
iterate over clusters and position nodes incrementally
update plot
```

4.3.5 Node-Link Circular Hybridization

Underlining the node-link graph of our visualization tool is a series of node positioning algorithms which ensure a clearer picture and reduce cluttering and edge intersections compared to, say, a random disposition.

While the main component of these algorithms is outsourced to the external library *pyGraphviz*, the positioning of isolated nodes is handled internally via a function which results in a ring-like placement around the connected graph(s). To be more precise, the algorithm requires two radii r and R and a center position (C_x, C_y) as parameters. It then computes, for each isolated node, a random number $0 < \theta < 360$ and a random number $r < z < R$, from which it determines the (x, y) coordinates of the node as

$$y = C_x + z \cdot cos\theta$$

and

$$x = C_y + z \cdot sin\theta.$$

That is, the locus of a ring with center (C_x, C_y), inner radius r, and outer radius R.

For simplicity, and given that the isolated nodes are only used as a visual estimate, no check is made for overlapping nodes. As for the parameters passed to the function, the two radii are calculated as a multiple of the initial zoom level of the *bokeh* plot, while the center point is set to the origin. Both of these choices are consistent with the way the main graph is constructed and positioned.

5 Application Example

Our example dataset consists of approximately 1,200 vertices and a total of 31,182 edges, changing dynamically over 1,077 timestamps, and we know that it is taken from the execution of some application. A hierarchy metadata file is also provided. As we open both files through the buttons at the top of the visualization, the tool immediately offers the three different approaches towards analyzing the dataset described in Section 4.2.

The dynamic timeline proposes an overview, allowing us to evaluate general patterns. When interacting with this visualization, we can select edges of interest and highlight them, while information about each edge is displayed by simply hovering over them. In Figure 3, we can immediately notice a dense mass of link crossings, which can be explored by filtering either by time step or node range: for example, focusing only on time step 1, we can observe that a great number of operations are executed here. The plot also reveals a second pattern which is much more regular, occupying most of the graph between nodes 300 and 800. We have enabled *datashader* for this analysis, which proves to be especially useful for this area, since the edges are so cluttered that they could be barely distinguished otherwise. Thanks to the *datashader* color scheme, which is proportional to the density of link crossings, we can see a repeating behavior between timestamps 350 and 950 for all nodes, with a small discontinuity around timestamp 600. Towards the end of the plot, we distinguish another pattern in which irregularities occur once again, although it seems like large chunks of the nodes are kept isolated.

With this information in mind, and knowing that the dynamic graph refers to the execution of some program, we can hypothesize that the application in question is **I-O intensive**: we have a first initialization which, as we would expect, is intensive in number of operations, after which the call of functions becomes sparse and linear. Upon request to close the application, once again the number of connections increase drastically, but as functions are returned entire chunks of nodes become isolated.

The adjacency matrix, as shown in Figure 4, also offers an overview of our data, but with a focus on the overall connections of the graph, rather than over time. One can easily notice that the matrix is very sparse, meaning that the number of isolated nodes is large in comparison to those connected. Plenty of points are located near the antidiagonal, suggesting that, given any node, it is likely it has an edge with its very close neighbors. We can also notice a slight horizontal line in the middle-left part of the matrix and, hovering, we can distinguish many edges starting from node 494. Otherwise, the adjacency matrix has a noise-like pattern, from which it could be difficult to infer other insights. However, once the adjacency matrix is ordered, as in Figure 7, it does show a few clusters, with two seemingly bigger clusters towards the two ends of the diagonal, and a couple of smaller ones in the central regions. This could be further inspected using the zooming and hovering interaction, but also by adjusting the time step or node ranges.

With the insights we already collected in the previous steps, we can move towards analyzing some specific time steps, in order to confirm our hypotheses. For example, we already stated that time step 1 contains most of the edges, but otherwise the graph is very sparse and has plenty of isolated nodes. Looking at the node-link diagram reinforces this idea: time step 1 from Figure 1 shows an extensive and somewhat hierarchical graph, where from a few central nodes several more nodes are reached, and the edges with greater weights are mostly on those central nodes (the largest cluster). The following time steps are not nearly as busy, most having no more than a couple of nodes and some having a dozen or so (the smaller clusters), and the number of isolated nodes is now even larger. The difference between these time steps can be seen by comparing time steps 293 and 425 in Figure 1. Within these visualizations, we can easily hover over specific nodes such as 624 or 357, since there are so few of them, and see that in the hierarchy file they seem to correspond to draw functions. Furthermore, the central time steps are almost all identical to each other, as we had already guessed from the timeline view. We can thus say that the program is a **graphical application** that is repeatedly drawing something on screen, and

from here it would be easy to debug outlier functions if needed. We can confirm that nodes like these are called repeatedly by using Dijkstra's algorithm on the dynamic timeline, obtaining a result similar to that of Figure 6.

6 Performances

Since one of the goals of our tool is **interactivity**, it is clear that we cannot allow for the three visualizations to take hours or even minutes to be generated. We set out to determine at which point the application would become unable to handle the given dataset.

Since our tool is web based, it has server-side operations, which include parsing, plotting, filtering, and applying algorithms, and client-side operations (managed by *bokeh*) which display the plots and GUI on the web page. While the former can be easily reconfigured to run on a more powerful server if need be, the same cannot be done about the latter, nor are we capable of optimizing its performances since it is handled internally by the library.

Furthermore, even if optimizations were possible, browsers are simply unable to guarantee responsiveness when the number of elements to display is too large. Fortunately, *bokeh* eases the latency in panning or zooming operations thanks to its **Level of Detail** (LOD) functionality, which temporarily reduces the number of elements on screen.

The following table contains the times the application needed to parse a dataset and execute its heaviest algorithms, in seconds. All tests were executed with server and client on the same machine, with an Intel i7 processor and 16 GB of RAM. Interactivity is highlighted by color, where a latency of less than 5 frames per second is indicated via an orange row. The datasets used for the tests are randomly generated, with about a 5% probability that an edge connects any two given nodes, and the number of nodes and time steps are parametric.

As it can be seen from Figure 8, the program only begins to slow down when a dataset of about 500,000 edges is loaded. At this stage the tool still operates, but with a latency that may ruin the user experience. For smaller datasets, the browser manages to visualize the data while maintaining a good performance; for larger datasets, the tool becomes unusable and does not display data properly, thus testing beyond this size has not been possible.

In terms of algorithmic operations, they tend to have trivial execution times, even for huge amounts of data, as thus they are not shown in the table. Notable exceptions are the application of a different layout for the node-link graph, which nonetheless keeps its execution time below the second, but most importantly the matrix ordering discussed in Section 4.3.4. Both versions of the algorithm present this latency and result in similar performances, either because the search for the shortest path or the computation of the average distance between clusters needs to be done once per iteration of the main loop of the algorithm, which has an expensive polynomial complexity $O(|E| \cdot |V|)$, where E is the set of edges and V the set of nodes of the input dataset. However, the *real* version performs slightly better, since it removes an element from the set of edges at each of its iterations.

7 Discussion, Limitations and Further Work

In this section, we will focus on discussing the main benefits, but also the drawbacks of using our tool, while also explaining the reasons behind our limitations.

We choose Python for its flexibility in syntax and great code readability, which we considered to be fundamental qualities to ensure an effective group work. Furthermore, the language provides a large number of easy-to-use libraries for data management, user interactivity, and data visualization, among which we chose *networkx*, *bokeh*, *pyGraphviz*, and *datashader*.

These allowed for quick and effective results, although we have lost control over some finer aspects of the application. A second bottleneck comes from potentially slower loading times in comparison to, say, a tool built in C or using a graphic library such as *OpenGL*.

Future revisions may concern two different key aspects, in terms of programming extra features. Firstly, practical improvements should be coded in order to make the program more accessible. Secondly, content-wise, many algorithms or further visual techniques that have been put aside in favor of those with higher priorities could be now implemented, building on our tool which is already a good framework.

Regarding the first point, it should be noted that while any machine capable of an internet connection can connect to a server running the tool and properly interact with it, the server itself can only display all three visualizations successfully in *Linux* and *MacOS*, due to an incompatibility of *Windows* with the *pyGraphviz* library used for the node-link layouts. Making the tool compatible with all major operative systems would be ideal for a more accessible program.

Another aspect that would increase the accessibility of our tool would be supporting more data formats since, so far, it only accepts the very specific format described in Section 3, which is not even the most commonplace format in use (which is *Comma-Separated Values*). Even more important would be a support for the most common types of metadata files, such that heterogeneous kinds of information can all be parsed properly.

Similarly, we may consider extending input compatibility beyond *Mouse* inputs, such that the tool allows as many different types of input as possible and still respond properly. Examples include *Keyboard*, which is the most obvious, but also advanced input systems that are becoming increasingly available, such as *Voice Commands* or even *Virtual Reality*. *Touchpad* may be the best candidate, since, as mentioned, the tool can already be looked at from a smartphone, provided a server is running the application.

Besides practical improvements, there is plenty of room for additional content in our visualization tool. Following are some suggestions of functionalities or options which can be adjusted, improved or simply added to the tool as it is right now.

The selection tool is fairly precise and shows some general information in the output section, but it can definitely be improved. Isolating single edges could be made easier and more detailed information could be given concerning a single edge, vertex or group, for instance. If this precise selection is accompanied by all possible information concerning that object, which can be achieved if support for more metadata is implemented first, the tool becomes as informative as it can be.

Beyond this, a functionality which implements a search by edges or nodes seems an obvious, and even essential addition to the tool, since it can provide clarity in situations where graphs can, at first sight, seem too cluttered to reveal any interesting property, and could even be paired with a re-configuring or reordering algorithm that focuses on the visualization on the searched elements and their relationship with all other ones.

A critical feature which is missing in our tool is direction: none of the layouts in Figure 1 show arrows or tapered edges that could distinguish the start node from the end node of a given link. This is due to limitations of *bokeh*: arrows could only be displayed as triangles, each of which would be treated as a separate object unlike the multi-line used to show all edges, and loading one element per edge would significantly slow down the application to the point that even for the example dataset discussed in Section 5 interactivity would have a latency worse than 5 frames per second on the test machine. As for tapered edges, HTML5 does not support it. However, it may be possible to develop a *Javascript* code that could supersede this behavior.

We worked on a functionality that should allow the user to create meaningful representations, although in the end it was too time-consuming to fully implement. The *datashader* library, used for

Timestamps	Nodes	Edges	Parse time	Matrix order (avg)	Matrix order (real)	Nodelink change
30	43	70	0.068	0.016	0.012	0.1599
57	48	121	0.106	0.011	0.021	0.215
42	94	209	0.096	0.161	0.134	0.262
93	97	489	0.187	0.191	0.145	0.336
500	200	9738	0.545	2.172	1.396	0.355
200	500	24741	0.945	39.663	23.966	0.439
500	500	61646	2.189	43.357	25.834	0.516
1000	1000	498547	14.237	429.983	293.71	0.574

Figure 8: Performance tests on randomly generated datasets, ran on an Intel i7 processor with 16 GB of RAM.

the timeline visualization, allows for proper GPU rendering calls which allow, for example, to change colors with a granularity of single pixels, rather than of *bokeh* elements. Comparing the timeline on the GUI in Figure 2 against Figure 3, the advantages of this render method become clear. Currently, this functionality only renders a static image, such that whenever the user pans or zooms the visualization does not update, but full interactivity with *bokeh* is possible. However, while our efforts in this direction have produced a working prototype, they are ruined by a nasty memory error in the underlying libraries, which is likely outside our control.

One last suggestion for further enhancement is the possibility to allow cross-highlighting between the three visualizations. That is, by selecting an element in one visualization, the user sees element is automatically selected in all visualizations. If this interaction will be implemented, it could also be extended to other cross interactions than just selection, like panning, zooming, or execution of algorithms.

All in all, there is still much room for improvement on our visualization tool: it works well for some users and use cases, but it could work perfectly for all of them. Nonetheless, the tool demonstrates to satisfy the five qualities we had hoped to achieve and expressed at the beginning of Section 4: it is compact, clean, and fast, even if its hierarchical nature is underdeveloped by the lack of a cross-highlighting feature. We also want to note that the suggested changes and extensions for a future version of the tool represent only a fraction of all possible improvements, since the field of dynamic visualization is in constant motion and there might be new ideas that we do not even imagine as of today.

REFERENCES

[1] D. Archambault, H. Purchase, and B. Pinaud. Animation, small multiples, and the effect of mental map preservation in dynamic graphs. *IEEE Transactions on Visualization and Computer Graphics*, 17(4):539–552, 2011.

[2] F. Beck, M. Burch, and S. Diehl. Matching application requirements with dynamic graph visualization profiles. In *17th International Conference on Information Visualisation, IV*, pp. 11–18, 2013.

[3] F. Beck, M. Burch, S. Diehl, and D. Weiskopf. A taxonomy and survey of dynamic graph visualization. In *Computer Graphics Forum*, vol. 36, pp. 133–159. Wiley Online Library, 2017.

[4] M. Behrisch, B. Bach, N. Henry Riche, T. Schreck, and J.-D. Fekete. Matrix reordering methods for table and network visualization. In *Computer Graphics Forum*, vol. 35, pp. 693–716. Wiley Online Library, 2016.

[5] J. Branke. Dynamic graph drawing. In *Drawing graphs*, pp. 228–246. Springer, 2001.

[6] V. Bruder, M. Hlawatsch, S. Frey, M. Burch, D. Weiskopf, and T. Ertl. Volume-based large dynamic graph analytics. In *2018 22nd International Conference Information Visualisation (IV)*, pp. 210–219. IEEE, 2018.

[7] M. Burch. The dynamic call graph matrix. In *Proceedings of the 9th International Symposium on Visual Information Communication and Interaction, VINCI*, pp. 1–8, 2016.

[8] M. Burch. Isoline-enhanced dynamic graph visualization. In *20th International Conference Information Visualisation, IV*, pp. 1–8, 2016.

[9] M. Burch, F. Beck, and D. Weiskopf. Radial edge splatting for visualizing dynamic directed graphs. In *GRAPP & IVAPP 2012: Proceedings of the International Conference on Computer Graphics Theory and Applications and International Conference on Information Visualization Theory and Applications*, pp. 603–612, 2012.

[10] M. Burch, S. Diehl, and P. Weißgerber. Eposee - A tool for visualizing software evolution. In *Proceedings of the 3rd International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT*, pp. 127–128, 2005.

[11] M. Burch, M. Hlawatsch, and D. Weiskopf. Visualizing a sequence of a thousand graphs (or even more). In *Computer Graphics Forum*, vol. 36, pp. 261–271. Wiley Online Library, 2017.

[12] M. Burch, M. Höferlin, and D. Weiskopf. Layered timeradartrees. In *15th International Conference on Information Visualisation, IV*, pp. 18–25, 2011.

[13] M. Burch, C. Müller, G. Reina, H. Schmauder, M. Greis, and D. Weiskopf. Visualizing dynamic call graphs. In *Proceedings of the Vision, Modeling, and Visualization Workshop*, pp. 207–214, 2012.

[14] A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012.

[15] P. Dewan. Words versus pictures: Leveraging the research on visual communication. *Partnership: the Canadian Journal of Library and Information Practice and Research*, 10(1), 2015.

[16] S. Diehl and C. Görg. Graphs, they are changing. In *International Symposium on Graph Drawing*, pp. 23–31. Springer, 2002.

[17] P. Eades, W. Lai, K. Misue, and K. Sugiyama. Preserving the mental map of a diagram. Technical report, Technical Report IIAS-RR-91-16E, Fujitsu Laboratories, 1991.

[18] M. Farrugia and A. Quigley. Effective temporal graph layout: A comparative study of animation versus static display methods. *Information Visualization*, 10(1):47–64, 2011.

[19] C. Friedrich and P. Eades. Graph drawing in motion. *J. Graph Algorithms Appl.*, 6(3):353–370, 2002.

[20] Y. Hu. Efficient, high-quality force-directed graph drawing. *Mathematica Journal*, 10(1):37–71, 2005.

[21] M. Kenney, J. Horgan, C. Horne, P. Vining, K. M. Carley, M. W. Bigrigg, M. Bloom, and K. Braddock. Organisational adaptation in an activist network: Social networks, leadership, and change in al-muhajiroun. *Applied ergonomics*, 44(5):739–747, 2013.

[22] J. Merrill, M. G. Orr, C. Y. Jeon, R. V. Wilson, J. Storrick, and K. M. Carley. Topology of local health officials' advice networks: Mind the gaps. *Journal of Public Health Management and Practice*, 18(6):602–608, 2012.

[23] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages & Computing*, 6(2):183–210, 1995.

[24] C. Perrow. *Normal Accidents*. Basic Books, 1984.

[25] M. Rahmes, K. Fox, J. Delay, and G. Roe. Matching social network biometrics using geo-analytical behavioral modeling. In *Computational Intelligence and Data Mining (CIDM), 2014 IEEE Symposium on*, pp. 422–430. IEEE, 2014.

[26] C. Vehlow, M. Burch, H. Schmauder, and D. Weiskopf. Radial layered matrix visualization of dynamic graphs. In *17th International Conference on Information Visualisation, IV*, pp. 51–58, 2013.