TU WIEN Informatics

Interactive Web-Based Visualization of Large Dynamic Graphs

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Matthias Reisacher, BSc
Matrikelnummer 1125358

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Silvia Miksch
Mitwirkung: Alessio Arleo, PhD

Wien, 13. Jänner 2020

Matthias Reisacher	Silvia Miksch

TU WIEN Informatics

Interactive Web-Based Visualization of Large Dynamic Graphs

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Media Informatics and Visual Computing

by

Matthias Reisacher, BSc
Registration Number 1125358

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Silvia Miksch
Assistance: Alessio Arleo, PhD

Vienna, 13th January, 2020

Matthias Reisacher	Silvia Miksch

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Matthias Reisacher, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 13. Jänner 2020

Matthias Reisacher

Kurzfassung

Die Visualisierung und Analyse großer Graphen spielt in verschiedenen Anwendungsbereichen eine wesentliche Rolle. Da die Größe von Diagrammen in den letzten Jahren exponentiell zugenommen hat, wurde es zu einer Herausforderung, das visuelle Durcheinander von dichten Graphen zu verringern. Durch die Abstraktion der Struktur eines ungerichteten Graphs, welcher Tausende von Knoten und Kanten beinhalten kann, wird das visuelle Durcheinander drastisch reduziert und die Analyse der zugrunde liegenden Muster in einem interaktiven Ansatz unterstützt. Zusätzliche visuelle Techniken werden verwendet um die Darstellung der strukturellen Entwicklung des Graphen und Beziehungen zwischen Entitäten in der Visualisierung zu überwinden. Die jüngsten Veröffentlichungen verwenden entweder "Rich Clients", die auf speziellen GPU-Algorithmen basieren, oder Cluster-Computing. Obwohl diese Techniken in der Lage sind, große Diagramme interaktiv zu verarbeiten, sind sie entweder durch die Hardware des Benutzers eingeschränkt, oder komplex und teurer als einfache Client-Server-Lösungen. Diese Arbeit zielt darauf ab, einen alternativen Ansatz zur Bereitstellung einer verteilten, plattformübergreifenden Server-Client-Anwendung bereitzustellen, mit der große Diagramme, die aus Tausenden von Elementen bestehen, interaktiv in einem modernen Webbrowser dargestellt werden können. Wir beschreiben eine auf Metaelementen basierende Aggregationsstrategie, die einen anpassbaren Detailgrad bietet und die Hierarchie kumulativer Elemente über mehrere Abstraktionsebenen hinweg visualisiert. Durch Hervorheben von strukturellen Änderungen in dynamischen Graphen über den Lauf der Zeit, in Kombination mit Techniken wie "Panning and Zoomingünd Överview and Detail", ermöglicht unser System die interaktive Analyse von Diagrammen. Wir werden die Verwendbarkeit unserer Technik anhand eines vollständigen Prototyps demonstrieren und die Geschwindigkeit anhand unterschiedlicher Testdaten messen. Darüber hinaus bewerten wir auch technische Aspekte unseres Ansatzes, sowie seine Anwendbarkeit auf große reale Diagramme.

Abstract

The visualization and analysis of large graphs plays an essential role in various application fields. Since the size of graphs grew exponentially in the past few years, it became a challenge to reduce the visual clutter of dense and occluded graphs. By abstracting the structure of a node-link diagram, containing thousands of nodes and edges, visual clutter is reduced drastically, supporting the analysis of underlying patterns in an interactive approach. Additional visual techniques are used to overcome the challenge of representing the evolution of structural diagram changes and relationships between entities in dynamic graph visualization. The recent publications of large static and dynamic graph visualization techniques are using rich clients based on fast processing GPU algorithms, as well as distributed approaches for cluster-computing frameworks. Even though these techniques are capable of processing large-scale graphs interactively, they are also restricted by the user's hardware or are more complex and expensive than simple client-server solutions. This thesis aims to provide an alternative approach, at providing a distributed, cross-platform, server-client application, able to visualize large node-link graphs, consisting of thousands of elements, interactively in a standard web-browser. We describe an aggregation strategy based on meta-elements, that provides an adjustable level of detail interface and visualizes the hierarchy of cumulative elements throughout multiple abstraction layers. By highlighting structural changes over time in dynamic graphs in combination with tools, such as panning and zooming and overview and detail, our system allows for dynamic graph exploration. We will demonstrate the usability of our technique by providing a complete prototype and present benchmarks on different graphs. Furthermore, we evaluate technical aspects of our approach as well as its applicability to large real-world graphs.

Contents

CHAPTER 1

Introduction

Graph drawing is an indispensable tool, due to the fact that many problems and domains can be modeled as graph structures [BF14]. They provide clues and trends in a visual matter, often accommodating large sets of data. In many areas of computer, social and natural sciences, relations between entries have to be identified and understood to gain more profound insights. Graphs can help the audience to grasp information visually, helping them to form hypotheses and efficiently draw conclusions by simplifying relational data [BJ76].

However, since the analysis of graphical structures depends on its visual representation, a visually appealing and well-structured layout is required. Therefore, several centralized and distributed layout algorithms have been developed in recent years to automatically reposition nodes depending on their adjacent edges, to enable visualizing a graph in an aesthetically-pleasing way [AA17, VLKS+11]. This procedure alone can be sufficient for small or medium graphs. However, node-link diagrams suffer from the problem, that the more information is visualized, the less meaningful they become due to the rising visual complexity. Therefore, large node-link diagrams consisting of thousands of nodes, cannot be rendered without significant overplotting, which makes them particularly challenging to read [BF14]. Thus, further abstraction methods have to be applied to simplify the structure of static graphs. Many techniques have been introduced to solve this problem, which can be classified into two major categories: aggregation of nodes and improvement of the edge layout [CW08, BF14]. In either way, the general goal is to produce summaries with controllable resolutions to simplify the structure [TY08].

A large class of real-world applications involves graphs that change over time [CT15]. For those dynamic graphs, visualizations aim to support the user by spotting changes in the overall network. The visualization of dynamic graphs has been recently a focus of research, and therefore, different classifications of visualization techniques have been

introduced [BF14]. Visualizing dynamic graphs is often done by animating a graph sequence, showing each part individually, or by displaying multiple snapshots of a graph side-by-side [CT15]. Nevertheless, the visualization of large dynamic graphs must deal with the same problems as the visualization of static graphs, to provide readable and appealing results which can be understood by a user. Additionally, the occurring changes of following graph states have to be visually communicated. To overcome this challenge, approaches became specialized to various application scenarios and alternatives to the animated node-link diagram, such as graphs plotted onto timelines, have been introduced [BF14].

Approaches based on a rich client, which executes the majority of calculations and processing tasks, are enhanced by implemented optimizations of fast processing GPU algorithms [ZM12, PA15]. Even though advanced general-purpose GPU programming enables highly parallelized processing, it also restricts the application by the user's hardware. Older machines with weaker hardware or mobile devices may do not have the required performance to visualize larger graphs. Additionally, all individual steps required to visualize a graph have to be executed by every user, even though they might visualize the same graph data.

On the other hand, applications based on frameworks for cluster-computing are able to avoid those restrictions. By pre-computing renderings of massive graphs and storing the resulting images on the server, clients can fetch those images to show a graph's visualization without the need of much performance [PA18]. But since computer clusters are more expensive and complex than a simple server-client setup, this approach might not be suitable for some application areas.

This work provides a distributed, web-based server-client application, able to visualize large static and dynamic node-link graphs, consisting of thousands of elements, in an ordinary web browser. Additionally, an already existing, state-of-the-art layout algorithm has been included in the infrastructure and overall workflow. Thus, creating a single solution capable of processing basic graph data in standard formats, calculating each element's position, creating summaries of the graph's layout by aggregating its components, and finally, visualizing the result. Hence, a network of micro-services will be created, enabling the application's backend to scale horizontally, making it adaptable to different workloads.

By using a combination of WebAssembly, a low-level assembly-like language with a compact binary format that runs with near-native performance [HRS+17a], and WebGL, a high-performance interactive graphics rendering API [Par12], the application allows interactive rates when using common graphics hardware. Meta node and meta edge aggregation for the visualization of graphs at different levels of detail, enable the user to examinate the graph structure at custom settings. Additionally, the tool contains panning and zooming at interactive rates, a detail view, a hierarchy visualization of

aggregated nodes, and the selection of individual nodes to show its attributes, allowing the user to explore the graph freely. Furthermore, dynamic information of a graph are visualized by color coding the lifetime of created and soon deleted nodes and edges.

1.1 Research Question

To the best of our knowledge, this application will be the first system to enable interactive visualization of large dynamic graphs in a conventional web browser without the need for a cluster of computers. Thus, the primary contribution of this thesis is to answer the following research question, by implementing and evaluating a prototype of the system:

What are practical implications of interactive visualizations of large dynamic graphs with modern web-based technology?

The main research question can be split into two detailed questions:

Q1 How can a large dynamic graph, consisting of thousands of elements, be visualized in a standard web browser?

Q2 How can real-time interactive exploration and visualization techniques be implemented with web-based technologies?

1.2 Methodology

The methodological approach includes following steps:

1. **Literature Review**

   The first part of this work consists of comprehensive literature research. Relevant scientific resources have been collected and analyzed. The gathered information serves as the theoretical base of the model and provides the context of this thesis.

2. **Technological Review**

   Information about current browser technologies working at a high level of performance is gathered. Thus, technical specifications and requirements are identified.

3. **Infrastructure Design**

   By analyzing previous approaches, possible solutions for a distributed system design and eventual bottlenecks are identified. Splitting individual tasks into separate micro-services enables a scalable solution while applying best practices of distributed systems to improve overall system performance.

4. **Prototype Development**

   A prototype of the system is implemented by applying the previously learned algorithms and technologies.

5. **Evaluation**

   The implemented prototype is evaluated by measuring the performance of individual parts of the system, as well as the processing time of the system for the main use cases.

1.3 Thesis Structure

The structure of the thesis is as follows:

2 **Graphs 101**: Provides necessary background information and concepts about different types of graphs and their properties.

3 **State of the Art**: Gives an overview of the topic and its many related fields, and identifies previously proposed approaches.

4 **Conceptual Overview**: Gives a systematic overview of the system's requirements, its components, and its concepts.

5 **Details**: Describes used technologies, frameworks, and implementation details of the working prototype.

6 **Results**: Contains the evaluation of the prototype and consists primarily of performance measurements.

CHAPTER 2

Graphs 101

As Bondy et al. already stated, "many real-world situations can conveniently be described by means of a diagram consisting of a set of points together with lines joining certain pairs of these points" [BJ76]. A graph forms a mathematical structure which enables the depiction of pairwise relations of objects, helping the viewer to gain actionable insights and to make data-driven decisions. Since the first paper about graph theory was published in 1736 by Leonhard Euler, graphs are used in almost every scientific field and especially in times of big data, their complexity and size have grown. Whereby the question of interest can often be phrased regarding structural properties of the graph. For example, the patterns of edges among vertex triples are of central interest in social dynamics studies, or the search for the shortest path between vertices form the base of movement information insights [Kol09]. Moreover, in many cases, temporal development can be observed. Visualizing such structures is a complex task, requiring prior knowledge about graphs in general. This section gives a brief overview of a graph's components and their technical terms, as well as a brief description of the different characteristics.

2.1 Graphs in General

During visualization, a perceptually efficient, structural representation of data or concept is created to support the decision making process of the viewer [War12]. Therefore, we have to discuss the general concept of *data* and *visual structures* first.

The classification of data is a big issue. One idea is to divide data into *entities* and *relations*. Entities are generally the objects of interest we wish to visualize while relations define the structures that relate entities to one another. Additionally, both entities and relations can have *attributes* which are used to assign various properties. Attribute values

can consist of multiple dimensions representing either nominal data, ordinal data, an interval or a ratio [War12].

Data can be represented by different graphical structures. Techniques for displaying general graphs can be divided into three main groups: *node-link based*, *matrix-based*, and *hybrid* [VLKS+11]. While all methods discussed in this paper are representing data via a node-link diagram, a description of alternative representation structures is given in 2.2.3: "Alternative visual representations of static graphs".

2.2 Static node-link diagrams

A node-link graph consists of a set of nodes, usually displayed as dots, circles or points, and a set of edges which connects a pair of nodes and is indicated by a line, joining the two points. A graph is called *finite* when both sets are finite. It should be mentioned, that this paper only includes finite graphs, so the term 'graph' always means 'finite graph'. There is no unique way of drawing a graph since the relative positions of vertices and edges have no significance [Kol09]. Nevertheless, some quality criteria for graphs like an even distribution of nodes and a minimal number of crossing edges are usually tried to achieve to generate clear graph diagrams. Different layout algorithms have been used to achieve this. For example, *force-directed layouts* simulate physical forces between nodes, a layout structure is applied in *hierarchical layouts*, or the plotting of edges is reduced to horizontal and vertical axes in *orthogonal layouts*. Those different layout types are shown in figure 2.1.

A *static* graph as

$$G := (V, E, \psi_G)$$

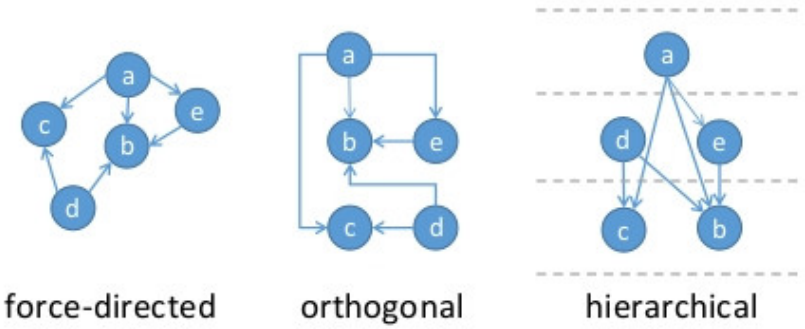


Figure 2.1: Different visual representations of static graphs showing the same dataset [BF14].

whereas V(G) models a non-empty set of *vertices*, E(G) a set of *edges* and ψ_G represents an *incidence function* that associates each edge of G a pair of (not necessarily distinct) vertices of G. So if u and v are both vertices of G, connected by an edge e such that $\psi_G(e) = uv$, than e is said to *join* u and v [BJ76].

2.2.1 Characteristics of static graphs

Since visualization algorithms are often adjusted to graphs with specific characteristics, a basic understanding of those properties is essential to be able to recognize the application area of an algorithm. Therefore, this section explains some of the most important characteristics of graphs.

Directed and undirected graphs

A graph can either be *undirected*, meaning that edges do not have a specific direction, or *directed*, in which case all links have an assigned orientation. Formally, a directed graph

$$D := (V(D), A(D), \psi_D)$$

with a non-empty set V(D) of vertices, a set A(D) of arcs and an incidence function ψ_D, associating each arc of D a pair of (not necessarily distinct) vertices of D. So if a is an arc joining the vertices u and v such that $\psi_D(a) = (u, v)$, then u is the *tail* of a, and v is its *head*. In opposite to an undirected graph, u and v are not interchangeable, since the order of an arc's ends is specified. Visually, a directed edge is represented by an arrow pointing towards the head of the corresponding arc [BJ76].

Weighted graphs

Each edge e of G can have attached a real number w(e), called *weight*. Those weights can be associated with different properties, depending on the respective application area. For example, the weight can be seen as the resistance of a wire segment in a physical network structure; or the weight can indicate the construction or maintenance costs of various links in a communication graph. Those weights are usually the subject of interest in various optimization problems, like the shortest path problem. Their goal is to find a subgraph of a particular type with minimum or maximum weight. A graph H is a subgraph of G (written $H \subset G$), if $V(H) \subset V(G)$ and $E(H) \subset E(G)$, with a corresponding incidence function $\psi_H(e)$. Whereas the weight w(H) of a subgraph H of a weighted graph G is defined as $\sum_{e \in E(H)}$ w(e) on its edges [BJ76].

2.2.2 Aesthetics

A readable, or aesthetic, graph provides easy access to detail information but also enables the viewer to detect general regularities and anomalies of the graph structure. To meet these criteria, static graphs should follow the following general aesthetic rules described by Beck et al. [BBD09]:

- *Visual clutter*, which describes a negative impact of excessive and/or disorganized elements on the performance on some task, should be reduced. For node-link diagrams, the total number of edge crossings should, therefore, be minimized.
- If similar visual elements representing different objects are positioned too close to each other, an effect called *spatial aliases* might appear. This effect can lead to perceptional mistakes where the viewer cannot distinguish between these objects. Spatial aliases can appear in node-link diagrams, for instance, when two edges are crossing at a small angle.
- The *compactness* of a graph, which describes how efficiently space (and time) is used to display information, should be maximized.

2.2.3 Alternative visual representations of static graphs

This section discusses some of the popular representational approaches, beside node-link based techniques.

Matrix representation

An alternating approach to visualize a graph is via an adjacent matrix, whereby rows and columns are used to represent vertices, and colored cells illustrate a connection with an edge for the two regarding nodes. Even though this approach does not have problems with crossing edges or overlapping nodes, which makes it suitable for dense graphs, it also suffers from scalability issues due to the limited display space, especially for very large graphs [VLKS+11].

Trees

Trees are visualized either on a node-link base, space-filling or a hybrid approach. *Space-filling* techniques are used to display hierarchy by using the full-screen area. Instead of links, the spatial position of nodes and there closeness or enclosure is used. The most

prominent techniques using enclosure are visualizing nodes as rectangular shapes in a recursive layout, where child nodes are positioned inside their parent nodes, subdividing their area. This approach allows an effective usage of the available space. While the heights and color of a node can be used to visualize arbitrary attributes, the size of a node is usually encoding the quantitative data [VLKS+11].

Network

In graph theory, a *network* refers to a directed weighted graph. For example, a flow network (also called transportation network) is a directed graph where each edge has a capacity value assigned in the form of a weight value. Each edge receives a *flow*, whereby the amount of flow cannot exceed the *capacity* of an edge. Those networks are used to model situations in which something travels through a network of nodes, like water in pipes or traffic in computer networks. However, the term network is not used consistently in literature. For instance, a social network not always refers to this specific structure [BF14].

A multivariate network N, which is a special type of a network, consists of an underlying graph G and n additional attributes $A := (A_1, A_2, ..., A_n)$ that are attached to the nodes and/or edges. An example of such a network is shown in figure 2.2, illustrating 421 documents with 55 attributes. The edge length is inversely proportional to the edge weight so that nodes are positioned closer together for larger weights [JI13].

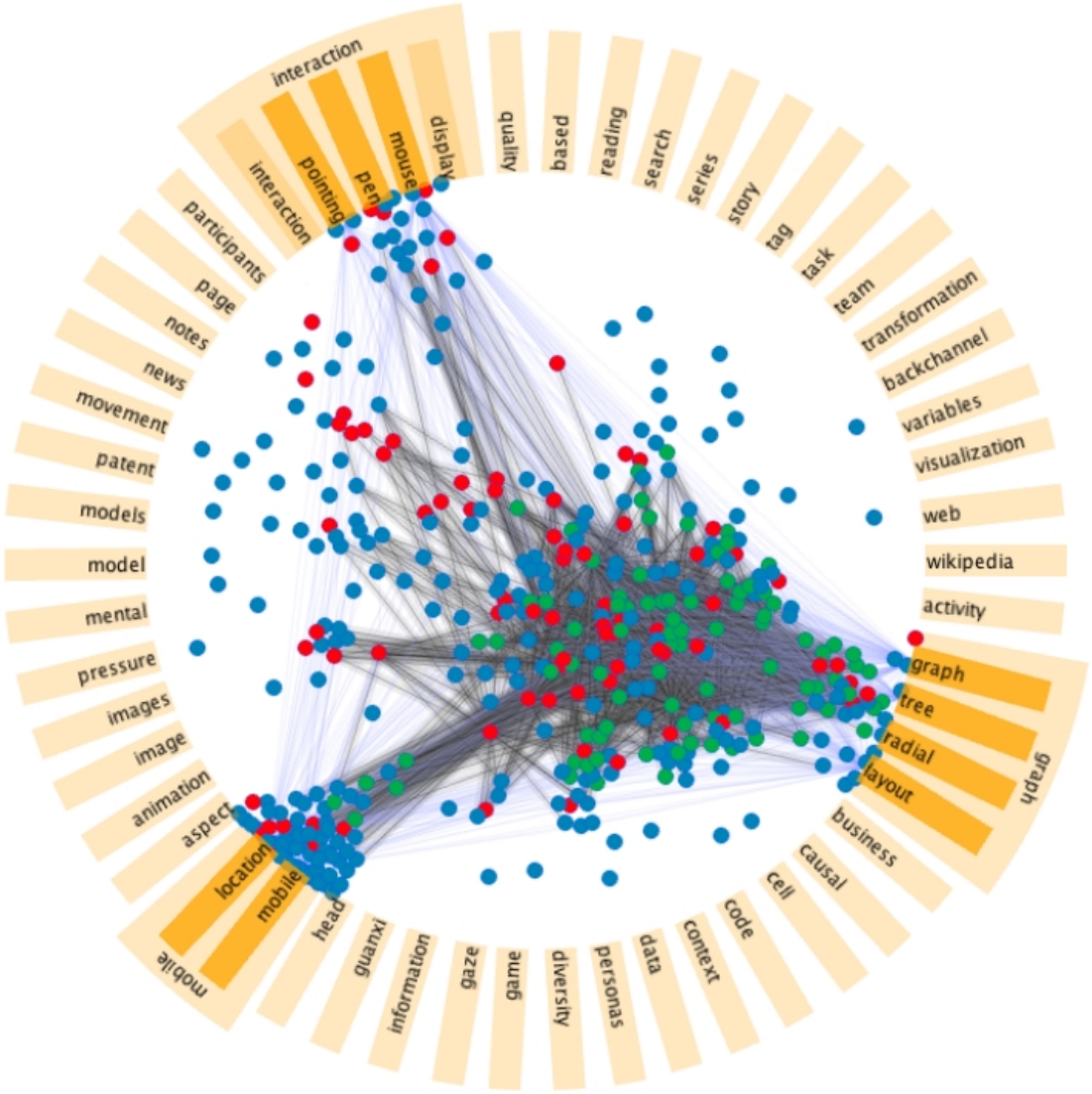


Figure 2.2: The screenshot displays a network of 421 nodes with 55 attributes. The nodes are colored differently, because they have been clustered based on their attribute values [JI13]

.

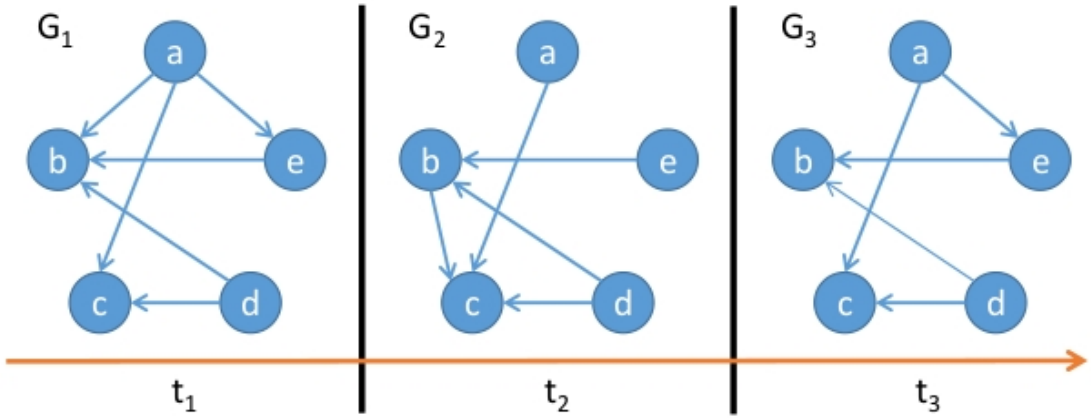


Figure 2.3: An example of a dynamic and directed node-link diagram on a timeline with constant nodes and updated edges [BF14].

2.3 Dynamic node-link diagrams

The characteristic difference of a *dynamic* graph to a *static* graph is the structural change over time. Dynamic graphs are subjects to discrete changes, whereby vertices or edges are inserted or removed. Figure 2.3 shows an exemplary visualization of a dynamic node-link diagram. Five nodes with a various number of edges are shown over a timeline of three steps. The nodes are not updated and remain constant to make the changes easier to track. For instance, an edge between node a and e exists at time step (t_1), disappears at time step (t_2) and reappears at time step (t_3) [BF14]. In general, dynamic graph problems can be classified according to the allowed types of updates:

- The unrestricted insertions and deletions of vertices and edges is called a *fully dynamic* graph problem.
- When only insertions of vertices and/or edges are allowed, the graph problem is called *partly dynamic* and *incremental*.
- And finally, when only deletions of vertices and/or edges are allowed, the graph problem is called *partly dynamic* and *decremental*.

A *dynamic* graph is defined as the sequence

$$\Gamma := (G_1, G_2, ..., G_n)$$

where $G_i := (V_i, E_i, \psi_{G_i})$ are static graphs of a sequence with the length n at the respective time steps $\tau := (t_1, t_2, ..., t_n)$.

As you can see, this data model considers time as being *discrete*. So continuous time scales must be represented indirectly, by mapping ordinal values to virtual points on a discrete time scale. In this case, *continuous* processes must be sampled to be represented in this data model.

Another concern is the differentiation between *instants* and *intervals*. G_i can either be a snapshot at t_i, representing the current state at a specific moment, or it aggregates an interval around t_i. Both representations are valid, and visualization techniques usually do not differentiate between those two models [BF14]. So it is entirely up to the data, how a visualized dynamic graph is mapped and therefore, what temporal resolution it provides.

2.3.1 Characteristics of dynamic graphs

The visualization of dynamic graphs has to address multiple challenges. On one side, the individual graphs must still be illustrated in a well-structured way, so the user can analyze it and gain new knowledge. Therefore, the same principles and algorithms of static graphs also apply for dynamic graphs. On the other side, the structural change over time must be communicated as well. This is especially difficult for large graphs, where nodes and edges are aggregated, and the limitation of space complicates this process. This section gives a summary of different requirements and limitations when visualizing dynamic graphs.

Animated approach

This strategy uses *time-to-time mapping* to project a sequence of graphs to an animated representation. The following graph states are shown in sequential order, one picture after the other. This procedure has the advantage of being able to use the whole screen space to show a single graph, which is especially crucial for large diagrams with thousands of nodes and edges. On the contrary, it can be difficult for the user to detect changes between two or more following graphs since no direct comparison of the individual stages is possible. Thus, the major challenge is to communicate areas of structural change to the user[BF14].

In layout algorithms for animated visualizations, requirements are discerned into the following two categories [HC14]:

- The *online methods* describes a situation where the individual node-link layouts are not known for future time steps, and therefore, only current data and data from

Figure 2.4: Different timeline approaches for a node-link diagram. On the left, the individual graphs are shown next to each other. Stacked 2D diagrams are shown in the right image. [BF14].

the past can be used for computation. Those methods can treat graphs as they become available.

- While the full evolution of the graph or graph sequence must be known at visualization time for *offline methods*, which usually simplifies the layout and visualization problem.

Timeline approach

The basic idea of *time-to-space mapping* is to show the complete sequence of graphs in a static image which helps to make changes traceable. However, the limitation of space can quickly decrease the readability of the diagram. To overcome the visual scalability problem, different approaches for positioning a graph are discussed. Figure 2.4 shows two examples of a timeline approach. The image on the left shows multiple graph stages placed next to each other, while the individual graphs are set on top of each other in the picture on the right. Other approaches suggest merging different graph stages and using a central timeline, but it seems that this technique is usually restricted to only particular types of dynamic graphs.

The visualization of *matrices* plays a particular role in time-to-space strategies. Since matrices are not very flexible concerning their layout, the challenge is to connect the spatial encoding of time within the matrix. Approaches have been discussed where the change over time is either encoded inside each cell by using various visual structures like a bar chart in combination with some color-coded schemata or visualized directly by layering multiple matrices on a timeline in different geometric shapes like a cube or a circle [BF14].

CHAPTER 3

State of the Art

3.1 Related Work

I found two state of the art reports during research, covering a related scope. This section describes their presented contents and results, focusing on the related matters. Additionally, a short discussion about open and missing topics will be provided.

3.1.1 Visual Analysis of Large Graphs: State-of-the-Art and Future Research Challenges

Von Landesberger et al. [VLKS+11] take a Visual Analytics perspective on the field of visual graph analysis by providing a systematic overview of the aspects of *visual representation*, *user interaction*, and *algorithmic analysis*. They have structured and analyzed their found papers in the following way:

- **Data transformation and mapping**

  In graph visualization, algorithmic graph preprocessing is used to simplify a graph to reduce its size while maintaining the main graph structure, as well as graph property modifications for repositioning or highlighting of nodes and edges. The two main approaches to graph reduction are *graph filtering*, where nodes and edges are removed from the graph, and *graph aggregation*, where nodes and edges are merged to single nodes and edges. Both approaches are keeping the underlying structure of the graph.

- **Visualization and user interaction**

  This section summarizes visualization and layout techniques for tree graphs, directed and undirected node-link diagrams, as well as compound graphs and matrix

representations, each on for static and dynamic data. Von Landesberger et al. emphasize the importance of the dynamic stability of the mental map in animated approaches.

Approaches for user interaction are categorized according to whether the user actions are applied on the *data values*, the *visual representation* or the *view*. They analyze interaction techniques including data filtering, visual abstraction, changing parameters and data values, and editing and aggregation.

- **Analysis**

  Analysis techniques can be applied in different stages of the visual graph analysis process. Structural analysis has a focus on the identification of essential nodes and connection between nodes, as well as substructures and the impact of graph changes on the structural properties. Another specific task is the examination of similarities and differences between different graphs, usually regarding their structure.

Von Landesberger et al. concluded an increasing interest in large graphs, which requires faster layout algorithms, as well as graph filtering and aggregation to maintain a readable diagram. Furthermore, dynamic and compound graphs have been the focus of research in the prior years, whereby the understanding of the graph changes must be supported by stable layouts that preserve the mental map. Especially dynamic approaches regarding the online problem, where data changes are not known, are a challenge in visual analytics.

Von Landesberger et al. give an comprehensive summary of many application areas regarding different graph types and scenarios, from preprocessing over visualization to user interaction techniques.

3.1.2 The State of the Art in Visualizing Dynamic Graphs

Beck et al. [BF14] collected and classified 60 publications as technique paper, concentrating on the two basic graph structures: *node diagrams* and *adjacent matrices*. They based their taxonomy on three major categories:

- **Time-to-time mapping**

  They found exclusively node-link diagrams using this strategy, which shows a sequence of graphs in an animated representation. Since each graph of the sequence is shown individually, and the user cannot compare individual stages, the major challenge is to communicate areas of structural change to the user. An important concept, which is discussed in almost every paper analyzed, is the *mental map*. It describes the structural information a user forms by looking onto the graph layout and should change only sparsely to keep the so-called dynamic stability, which

helps the user to spot changes more easily. Therefore, visualization is done by abstracting individual structures or minimizing the number of changes per frame to make changes more traceable by the user.

- **Time-to-space mapping**

  This section describes different variations of timeline approaches. They found techniques for node-link diagrams positioned next to each other, preferably applying a fixed layout of the nodes, which is called *juxtaposition*, stacked on top of each other, which is called *superimposition*; or the sequence of graphs can be merged into an integrated diagram in a so-called *integrated* form.

  Matrix-based visualization approaches, on the other hand, are described in two categories. With *intra-cell timelines*, each cell contains an individual timeline to represent the dynamic changes of the edge, whereby different forms of such representations exist. For instance, temporal data can be visualized by a bar chart or by stacked lines within a single cell, or by simple color-coding. *Layered matrices* describe a category where adjacent matrices are juxtaposed or layered onto a timeline in a radial form or a circle.

- **Hybrid**

  Hybrid strategies use both representations closely connected. Different techniques like Parallel Edge Splatting are used to embed animated diagrams into a timeline.

During their evaluations, they found out that the importance of the mental map was not evident. Some papers found minor improvements by preserving the user's mental map, and others could not detect differences at all, indicating that its role might have been overestimated in literature.

As for comparing animated with timeline approaches in regards to the question of superiority, it seems that timeline-based procedures should be preferred for tasks involving more than two-time steps. For other tasks, studies could not provide a clear picture.

Research about dynamic graph visualization techniques started in the 1990s. Since then, the number of publications rose, and the approaches became more and more specialized. The paper introduced by Beck et al. is intended to give a broad overview of the released papers since then. Thus, the focus lies more on the overall strategy and classification than on the applied technique of the individual papers.

Even though Beck et al. are giving a comprehensive overview of the general topic, they are discussing almost exclusively layout algorithms for dynamic graphs and are missing the connection to static graph visualization. Especially with large graphs, preprocessing and aggregation techniques have already been used to create understandable diagrams, and color-coded edge bundling methods have been used to highlight areas of change in large-scale graphs.

3.2 Method

3.2.1 Scope

The specific scope of this thesis is visualization methods for node-link diagrams in a general-purpose layout, which are based on or extensible to time-to-time and time-to-space mapping techniques. Many of the existing techniques are surveyed by Van Landesberger et al. [VLKS+11] in the year 2011 and Beck et al. [BF14] in the year 2014. Landsberger et al. discuss the field of visual graph analysis in a broader sense, summarizing mostly static visualizations of different kind of graphs with various user interaction techniques, while Beck et al. present the visualization of dynamic graphs in different categories, focusing on different layout algorithms.

As Keim et al. [KAF+08]. stated, the Visual Analytics process consists of the following four key steps, whereby each part effects the final visualization:

- Data pre-processing
- Layout selection
- Visualization
- Visual user interaction

Interaction can be applied to support users by solving tasks connected to the exploration of graphs [VLKS+11]. Some of the existing interaction techniques have already been surveyed by Van Landesberger et al. [VLKS+11] and Beck et al. [BF14]. Graph layout algorithms can be used to visualize the change in data over time, but there are more specialized visualization techniques for this problem. Especially with large graphs, layout algorithms tend to rely on computation networks and parallel processing, due to the sheer amount of data which must be processed [AA17, HA15]. Hence, I considered layout algorithms and interaction techniques out of the scope.

Furthermore, to avoid re-discussing techniques already analyzed by those two surveys, most of the papers included in this chapter have been proposed in the year 2014 or later.

3.2.2 Data Collection

Collecting relevant papers for this survey, I started with a selection of papers that I got from the supervisor of my master's theses. I followed citations in both directions, checked the list of references in the papers to find previous works and investigated them using Google Scholar. But only papers inserted by default peer-review, entirely written in English and published in journals and conferences were added to the scope of this review paper.

Some journals and conferences of interest:

Journals:

- Computer graphics forum
- Information Sciences

Conferences:

- IEEE Transactions on Visualization and Computer Graphics
- IEEE Transactions on big data
- ACM International Symposium on Visual Information Communication and Interaction
- ACM International Conference on Management of Data
- ACM Conference on Human Factors in Computing Systems
- Symposium on Graph Drawing and Network Visualization

Keywords:

static graphs, dynamic graphs, aggregation techniques, visualization, node-link, edge bundling

The retrieved dataset after applying the described methodology consists of 11 papers, whereby four articles have been neglected and categorized as out of scope. The remaining papers have been ordered by their aggregation technique and sorted by the performance of the individually proposed algorithm. The two significant categories of aggregation techniques are *edge aggregation* algorithms, as well as *edge and node aggregation* algorithms. It should be noted, that no categorical differences were made between distinct strategy levels such as data-based, geometry-based or image-based methods [PV15].

3.2.3 Out of Scope

The following papers have been excluded from the scope:

Visualizing a sequence of a thousand graphs (or even more)

Burch et al. [BM17] proposed a method to visualize a whole sequence of a dynamic graph in one static representation by using a parallel edge splatting technique. This enables the user to gain insights into the dynamic behavior of related entities. But since this approach represents an alternative visualization technique to time-to-time and time-to-space mapping, this paper was excluded from the scope.

Nanocubes for Real-Time Exploration of Spatiotemporal Datasets

Lens et al. [LL13] proposed a real-time visualization technique for large multidimensional spatiotemporal datasets to analyze spatially or temporally correlations between node attributes. To accomplish this, a data structure called *nanocubes* has been introduced, which represents data cube aggregation operation and contains all possible precomputed aggregation querries over the database. Since the focus of this approach is mainly on the nanocube data structure, which depends heavily on the memory, the paper has been excluded from the scope.

Animated Edge Textures in Node-Link Diagrams: a Design Space and Initial Evaluation

Romat et al. [RH13] proposed an approach to visualize data attributes of graphs in contrast to previous methods, not with color, opacity, stroke thickness or stroke pattern. Instead, they use animated edge textures to create playful and aesthetic visualizations. Even though, the technique lacks an advanced aggregation technique to a successfully applied to medium or larger graphs.

Visualizing Dynamic Weighted Digraphs with Partial Links

Schmauder et al. [SH15] proposed a technique to visualize time-varying weighted node-link diagrams by aggregating links and exploiting them as timelines. But since weighted graphs are in the focus of this paper and the technique relies heavily on the underlying layout of the diagram, this paper has been excluded from this papers' scope.

3.3 Results

The papers collected and analyzed during literature research are discussed in detail in this chapter to give a profound insight into different strategies and approaches for static and dynamic graph visualization. The individual papers have been classified into two distinct categories, *edge aggregation methods* and *node and edge aggregation methods*.

3.3.1 Edge Aggregation Techniques

The overall goal in edge aggregation methods is to merge single edges to reduce the size of the graph, while simultaneously revealing underlying patterns in the graph structure. There are various ways to aggregate the edges of a graph which can be grouped into three basic strategies [PA15]:

- **Data-based methods**. Clutter is reduced by filtering edges, showing a simpler graph structure as a result. Papers of this strategy have already been discussed by Panagiotidis et al. [PA15] and since no newer papers of this kind of approach were found during research, the discussed papers are all associated with the following two strategies.

- **Geometry-based methods**. Edges are grouped by bundling techniques, based on similarities in attributes like distance in the graph *drawing*. The selected edges are deformed to overlap to increase separating whitespace and to reduce the clutter.
- **Image-based methods**. Those algorithms have been optimized to exploit the parallel computing power of GPUs while using color and opacity interpolation to visualize different attributes.

The following papers give a detailed overview of the different strategies and their implementations which were proposed in the recent years.

Geometry-Based Edge Clustering for Graph Visualization

Weiwei et al. [WC08] proposed an edge-clustering technique based on control meshes to reduce visual clutter and enhance patterns in graphs. Their method is inspired by road maps, where connections are no longer straight lines between nodes, but rather segments which consist of cities and highways. So the goal is to turn general straight line graphs into road-map-style graphs to effectively reduce clutter, which helps to detect a high-level pattern in the data.

The approach consists of three steps: control mesh generation, edge clustering, and visualization. The basic idea is first to generate a control mesh, which consists of triangle meshes, that reflects the underlying edge distribution. Figure 3.1 (a) shows an example of a graph with its corresponding control mesh. Next, control points are created on edge clusters to bundle edges with similar direction and length, which is done by intersecting the control mesh with the original edges and calculating control point on the control mash edges for each intersection cluster. Figure 3.1 (b) shows the intersections between the links as red dots and the corresponding control point as red crosses. Afterward, a K-means clustering technique is used to calculate one or more control points for each line.

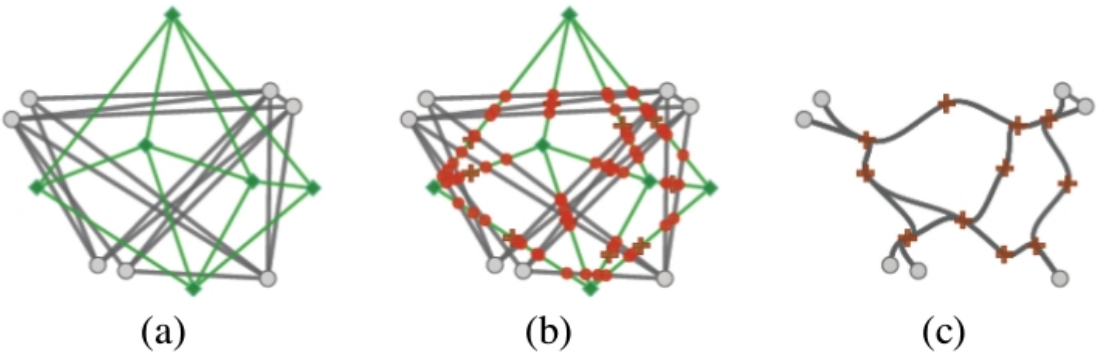


Figure 3.1: Edge clustering by control points: (a) a graph with a control mesh; (b) the intersections and the control points; (c) the merged graph.

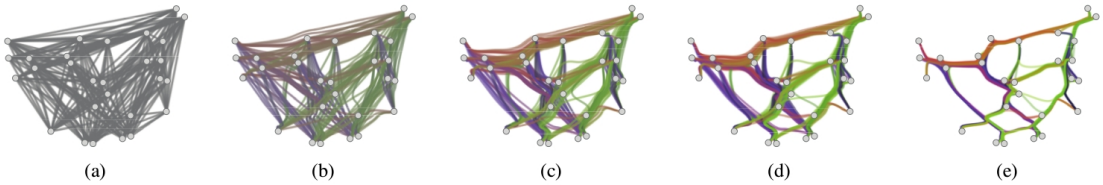


Figure 3.2: An animation sequence for an edge-clustering process. The color is used to encode the edge directions [WC08].

The edge bundler forces the edges to pass through their assigned control points to create an edge-clustered graph. Because each line becomes a polyline in the final layout, a local smoothing algorithm is applied to turn zigzag lines into curved edges. This is visually more pleasing and prevents the indication of a wrong edge direction. The resulting layout is shown in figure 3.1 (c). Finally, color and opacity techniques are applied to enhance different patterns in the graph. By computing the line density, the average distance and the direction of each edge bundle, a transfer function maps these attributes to color and opacity values.

Figure 3.2 shows some frames of an animation sequence, which shows the transition from a straight line graph to a color-coded edge-clustered diagram.

Weiwei et al. tested their approach on several graphs of different types and structures. No edges are removed from the graph display, underlying graph patterns and the context are enhanced, helping the user gaining profound insights. Since the whole method is built around the concept of generating control meshes to route the bundling, the final result relies highly on the quality of control meshes. Therefore, next to the automated mesh generation algorithm, the framework provides features which allow the user to manipulate the control mesh. But if the graph data does not contain clusters of edges with similar direction, the approach does not work at all, and the user must switch back to the original graph.

Bundled visualization of dynamic graph and trail data

Hurter et al. [HC14] have introduced a GPU-based edge bundling technique to simplify the connectivity patterns of large graphs. The overall goal is to put adjacent edges in tight bundles while preserving the spatial and temporal continuity. In other words, graph structures which are stable in time should also be stable in the animation, while on the other hand, substantial changes in the graph should lead to a visible difference in the animation. However, even massive structural changes should not lead to discontinuous bundle jumps, since such sudden changes aggravate visual tracking.

Their approach is suitable for static and dynamic large graphs and is called *kernel-density estimation edge bundling* (KDEEB). It estimates the spatial edge density with an Epanechnikov kernel to create a density map, which is computed by splatting the kernel, stored as an OpenGL texture, into a 2D buffer. Afterward, they move all edge points into bundles by applying 5 to 10 Euler iterations to create the final visualization of a static graph. So basically, KDEEB is nothing else but the well-known mean-shift algorithm [PV15].

For streaming graphs, those iterations are not executed on a single timestep. Instead, each keyframe is statically bundled, and edge correspondences are linearly interpolated in-between. Thus, they calculate the bundles on the fly while advancing in time to the next keyframe. This approach has the advantage, that when the graph changes slowly, the result is almost similar to the one of a static diagram, but with an improved overall performance. On the other hand, the algorithm has less time when the graph changes rapidly, which leads to looser bundles, representing the dynamics of the graph accordingly. Additionally, new edges and disappearing edges are specially handled to help the user find change over time or deviations from regular patterns in the animation. While newly created edges are bundled as times goes by, disappearing edges are interpolated from their bundled position to their original location. To emphasize this effect, color blending is used to change an edges' transparency correspondingly by applying a respective fade in or fade out effect.

Figure 3.3 shows the results of a dynamic graph represented as a sequence visualization of six consecutive days. You can see, that the same time-of-day flight patterns are quite similar for several days, but differ strongly over a day:

- Most of the traffic is concentrated on the East coast during the evening.
- During the afternoon, the traffic distributes uniformly across the whole US.
- Flights linking the East coast and the West coast dominate during the night.

Hurter et al. have proposed a robust and simple to use method for static and dynamic general graphs, which updates the graph *incrementally during* the bundling. They evaluated their algorithm manly by comparing the results and the performance against similar algorithms, whereby their method outperformed previously introduced approaches by a factor of up to 10. But they had not administered a quantitative or qualitative study to prove the effectiveness of animated bundles in general.

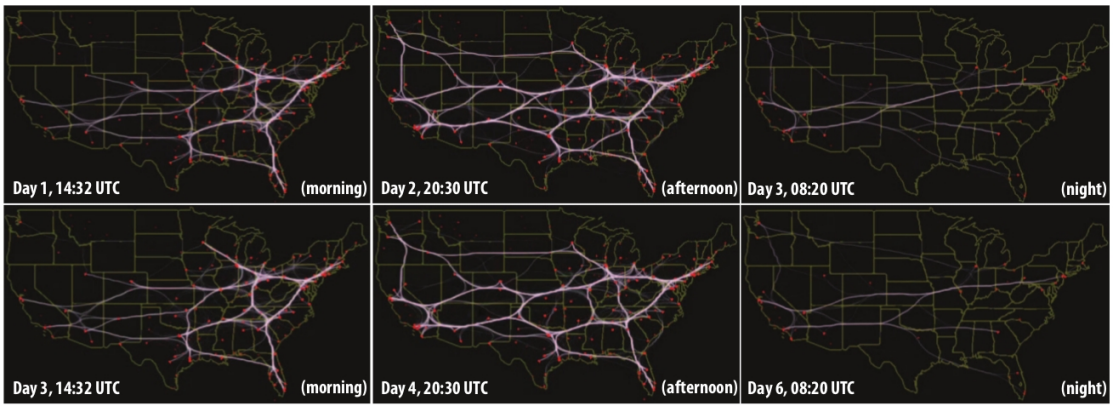


Figure 3.3: Streaming visualization for 6-days US airline flight dataset (41K flights) [HC14].

Attribute-Driven Edge Bundling for General Graphs with Applications in Trail Analysis

Peysakhovich et al. [PV15] presented a technique to generate bundled graph layouts according to any numerical edge attribute such as direction, timestamps or weight. This ability makes the framework suitable for trail visualization, allowing users to execute tasks such as exploration of large trajectory datasets. In such *attributed graphs*, the trajectory endpoints are represented by nodes and the actual path of the motion is represented by (curved) edge control points.

Their algorithm is called ADEB, which stands for *attribute-driven edge bundling*, and can be controlled flexibly to include (mixes of) edge attributes. The overall approach is very similar to the KDEEB [HC14] algorithm, which is basically a mean-shift algorithm. But in contrast to KDEEB, the density map, which is calculated on the GPU and stored as a texture, is estimated now only over the subspace of compatible directions.

Additionally, a directional colormap is used to show the direction of each bundle, whereby a color wheel is used to communicate, which color indicates which direction to the user. Furthermore, the luminance is increased linearly along edges to help the user understand the color mapping more easily. This way, the direction is also shown by following the dark-to-saturated color gradient.

Figure 3.4 shows the results of the aircraft trajectory dataset. While image (a) shows, the original data, image (b) shows the directionally bundled, color-coded and luminance bundled trails. Finally, image (c) shows the zoomed-in visualization of flights in a circle of about 100 km around Paris. As you can see, this flow pattern has only four main incoming and four main outgoing flows, marked in the image (c) by arrows.

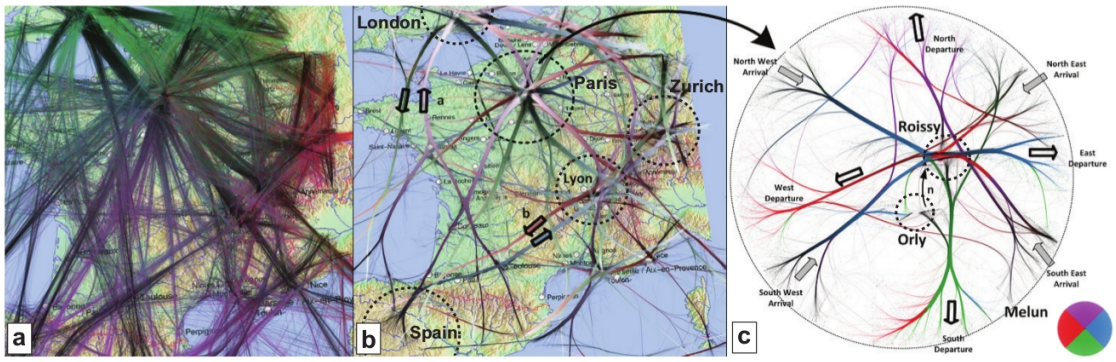


Figure 3.4: Analysis of aircraft trails over France. (a) Raw data. (b) Directional bundling. (c) Zoom-in over Paris area [PV15].

They tested and analyzed their method with two known data sets, the aircraft trajectory datasets and eye-movement traces, and two multitask experiments in collaboration with the French Civil Aviation School. The multitask experiments datasets where created with eye tracking, while the subjects performed specific tasks during the priority management test. Overall, the ADEB algorithm created structured bundles allowing in-depth analysis. However, strongly cluttered raw trails data prevented them from seeing high-level scan path patterns.

Performance wise, the KDEEB algorithm for undirected data is roughly twice as fast as the ADEB algorithm. This is attributed to the additional costs of computing the flow direction map.

Since the algorithm maps distances of attribute-values to angle distances, graphs with ordinal and quantitative values can be visualized. But edges attached with categorical values can only be visualized individually. The bundling of transitions between categories would require an algebraic definition of the category space.

CUBu: Universal Real-Time Bundling for Large Graphs

Van der Zwan et al. [VdZM16] proposed an entirely GPU-based real-time edge bundling algorithm, which can process large graphs consisting of up to a million edges at interactive framerates. The overall goal is to group spatially close and semantically related edges to reduce edge crossing clutter and show the graph's underlying pattern.

Their algorithm is called CUBu (CUDA-based Universal Bundling) and is implemented in C++ and NVidia CUDA 2.1. The bundling algorithm is based on the KDEEB method [HC14], which works with a mean shift technique, improving the overall performance by optimizing the approach regarding GPU parallelization and fixing accuracy issues.

The original *density map* computation was implemented by a 2D convolution, which scatters pixel values to neighboring pixels inside a specific kernel radius. Since this

approach suffers from many concurrent image-writes, the convolution was split into two 1D passes, whereby one is executed over the rows and one over the columns of an image. This approach has the advantage that several blocks can be processed in parallel via CUDA kernel invocation. Furthermore, the scattering strategy of the kernel was replaced with a *gathering* strategy, which allows the usage of the *atomicAdd* operator to take care of concurrent writes.

Another improvement of the KDEEB algorithms is the adaption of the *advection and resampling* step to move edges upwards in the density gradient. By applying mathematical projections, the resampling of edges can be done after every third or fourth iteration, instead of every iteration, improving performance while producing more accurate results. This, on the other hand, allows decreasing the number of computations needed for smoothing the edges.

Additionally, they have implemented multiple different visualization styles for bundled edges. Figure 3.5 shows a comparison between previous approaches and new techniques implemented in CUBu, based on the edge length and direction. As you can see, the CUBu visualizations show many more detail edges connected to isolated nodes.

They have tested the CUBu algorithm on several large real-world graphs, such as flight exploration and eye tracking, regarding speed and quality of the resulting visualizations. Their main focus was on the following criteria:

- *Scalability*: The algorithm's ability to work as fast as possible.
- *Directions*: Processing of directed and undirected graphs.
- *Level of detail*: Processing of directed and undirected graphs.
- *Generality*: Extendability of the algorithm to solve various problems with specific constraints.

The COBu algorithm is on average about 50 to 100 times faster than the KDEEB algorithm, the fastest algorithm for undirected graphs by then, and about 60 to 200 times faster than the ADEB [PV15] algorithm, the fastest algorithm for directed graphs by then.

The COBu algorithm can visualize edge bundles in all the styles shown in figure 3.5, which helps the user to get more profound insights by choosing an appropriate visualization for a specific dataset. Since the algorithm works in real-time, the user can see the results instantly.

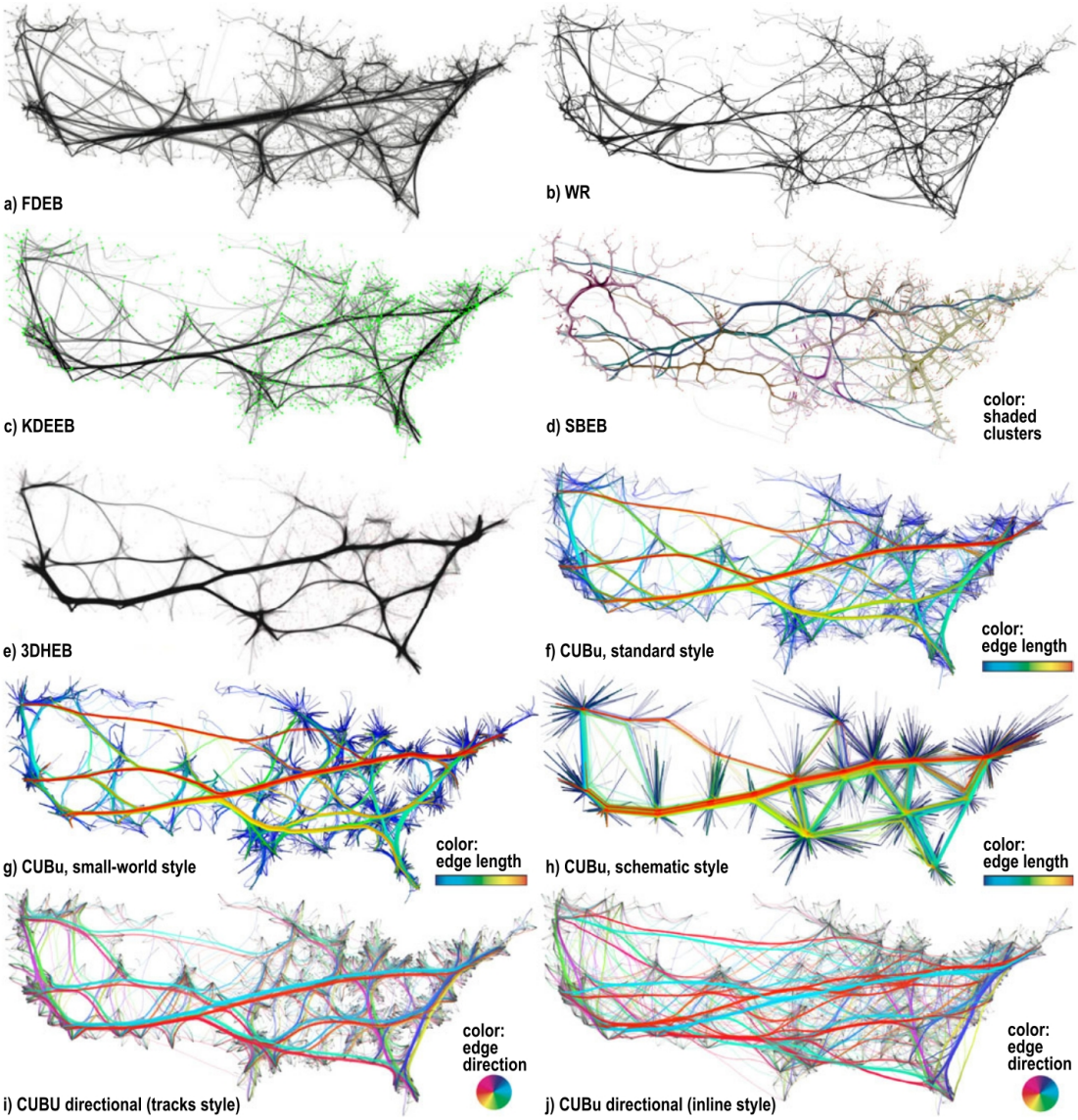


Figure 3.5: Bundling styles for *migration* graphs, comparing existing algorithms (a-e) with styles produces by the COBu method (f-j) [VdZM16].

3.3.2 Node and Edge Aggregation Techniques

Graph aggregation and filtering are the main strategies to simplify graphs and ultimately to draw them in a meaningful way. The aggregation of nodes has the main advantage to illustrate groups of entities. Nodes are clustered by either mutual attribute properties or structural proximity. But usually, some edge aggregation is applied additionally, since the number of edges often exceed the total number of nodes. Thus, not aggregated edges are often leading to cluttered graph visualizations. By combining both aggregation techniques, groups of entities and the relationship between them can be revealed.

A popular way to enable the adjustment of individual approaches to specific scenarios and use cases is by coupling them together with interaction techniques like semantic zooming, with the goal to increase the level of detail by going down to lower levels of aggregation. Another favored way of changing the aggregation level by user interaction is to improve the data values. The user manipulates the data directly to add and remove aggregated nodes on demand.

The following papers give a detailed overview of the directions in the past years.

Interactive Level-of-Detail Rendering of Large Graphs

Zinsmaier et al. [ZM12] presented a method for node *and* edge aggregation of large-scaled graphs to reduce the overall cluttering, allowing efficient rendering in different levels of detail at interactive rates. This is achieved by using a heatmap-like visualization by grouping adjacent nodes and edges which is done exclusively on the GPU. The implementation of their method is called *large graph observer* (LaGO) and can render graphs up to the size of $\sim 10^6$ edges below one second, while still providing zoom and pan operations to support the interactive exploration.

The basic idea is to aggregate nodes by creating density fields with a *kernel density estimation* (KDE) method. The 2D kernel function is precomputed and stored as an approximating floating point texture. A triangle filled with this texture is then created for *each* node. These triangles are accumulated with additive blending in the rasterization pipeline creating the visualization of the aggregated nodes.

But since this method is inefficient for large graphs, the total number of nodes is reduced priorly by binning nearby node coordinates into pixels. To achieve this efficiently, Zinsmaier et al. proposed a technique called **Seed Point Method**, consisting of two render-passes. The first pass creates an *aggregation field*, where nodes are merged into pixel-sized bins using a frame buffer object in screen resolution. The second render-pass applies the weighted triangles containing the kernel texture for every bin using a geometry shader. This approach optimizes the input set for a given viewport, by fitting neatly into the OpenGL pipeline.

Since nodes are aggregated and clustered together, the corresponding edges must be moved as well to visually couple edge aggregation and node aggregation. The aggregation

Figure 3.6: US census data set with 545,882 weighted edges depicting the county-to-county migration flow between 1995 and 2000 [ZM12].

field, which was created during the first pass of the node aggregation step, is interpreted as a height field. By using a hill-climbing algorithm, the starting and end points of the edges are moved to the *local maxima* of this field. This approach leads to the removal of inner-cluster connections and emphasizes intra-cluster connections, further reducing the total amount of edges and clearing the clutter. The resulting edge aggregation texture contains the total number of edges passing through for each pixel. Finally, the aggregated edge texture is processed by color-mapping the texture and applying an asymmetric pseudo-Euclidian distance function to render the various thickness of edges.

Figure 3.6 shows a visualization of the US census dataset with LaGO. The zoomed perspective (right lower corner) allows the user to see connections between individual counties while still being able to observe higher level patterns in the aggregated overview.

Zinsmaier et al. have tested their approach on three known data sets: US air-traffic, US migration, Net 50 and OpenStreetMap data of Europe. They have concentrated primarily on the performance of the LaGO algorithm and the overall quality of the created visualization in combination with the interactive tools. Additionally, a random graph with 200,000 nodes and 100,000 edges has been used to validate the representation regarding a quasi-equal distribution. Overall, no apparent patterns occurred, meaning that the algorithm avoids false-positive graph patterns.

Consistently GPU-Accelerated Graph Visualization

Panagiotidis et al. [PA15] proposed a method to visualize a graph using the GPU's graphics pipeline exclusively. They are using a data model specialized for the topology of modern GPUs to optimize their graph layout and rendering algorithms, as well as user interaction.

The basic idea is to perform every operation in-memory on the GPU, either as a shader or a rendering program. Thus, the data is preprocessed into their data structure on the CPU first and transferred into various shader storage buffer objects on the GPU afterward. This strategy minimizes expensive read back operations to the host, boosting performance. Instead of storing information about a graph in a square matrix structure, they have separated the data into three parts - edges, nodes, and layout links - linking them together via self-assigned IDs.

When the data structure has been transferred, the graph is iteratively processed by multiple shader programs executed in consecutive passes. First, bounding boxes for nodes and links are created to support non-overlapping nodes. Afterward, a *force-directed layout* algorithm based on the Fruchtermann-Reingold method is processed, whereby repelling and attracting forces are calculated in parallel.

When the final element positions have been set, the rendering pipeline starts drawing links and then nodes on top. Thus, nodes and edges can even be distinguished when there is a large number of edges. For directed graphs, Panagiotidis et al. suggest illustrating the direction of edges by either drawing an arrowhead at the edge's tail or by using a color gradient. The edge weight can be indicated through opacity or thickness.

Various operations have been implemented to support interactive exploration. The user can navigate through the structure of a graph with *panning and zooming*. Nodes and links can be selected and moved, pined or hidden. Furthermore, the rendering process can be paused and configured at any time to preserve the user's mental map.

Figure 3.7 shows an example of their approach visualizing the migration dataset. To illustrate the normalized amount of overlapping edges and the edge weight, the color of links is mapped respectively using tone mapping.

Panagiotidis et al. have evaluated their approach by rendering protein structures and measuring the performance of the individual steps.

An advantage of their proposed method is the possibility to replace the traditional visualization of a node with an arbitrary geometric form or with an image, which might be a better option for particular illustrations.

Although, their method requires a large amount of GPU memory to store all the required information to render a graph. Their results showed a total memory requirement of more

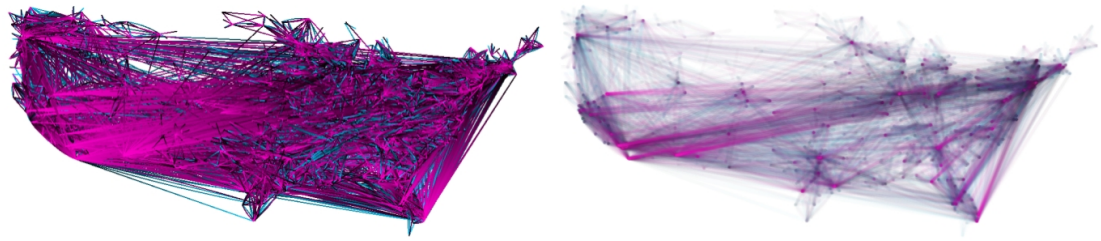


Figure 3.7: The US migration graph rendered directly (left) and with tone mapping using per-pixel fragment counters and floating point framebuffers (right) [PA15].

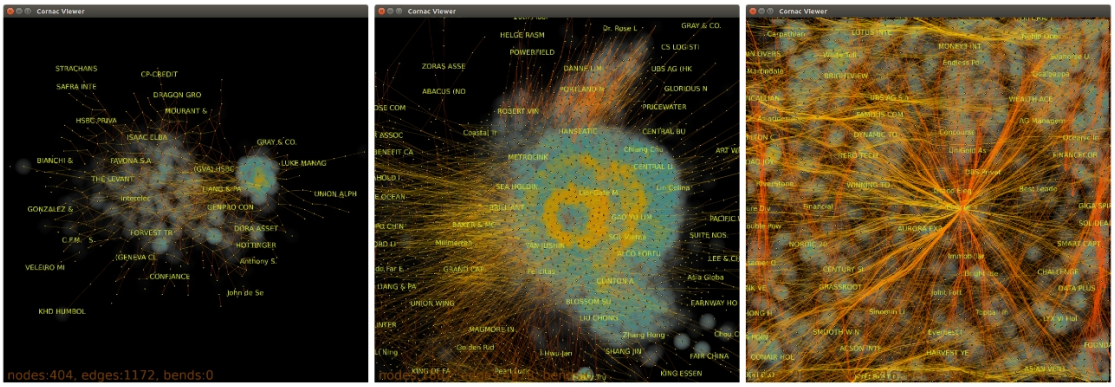
than 4 GB to process a graph with 16K nodes and 128M edges.

Cornac: Tackling Huge Graph Visualization with Big Data Infrastructure

Perrot et al. [PA18] presented a client-server approach to visualize and interactively explore huge, precomputed graphs by using a Big Data ecosystem and a light-weighted client running in a web browser. Their main focus is on data that big, that it cannot be stored inside centralized storage. Instead, solutions like the de facto standard *Hadoop* are using data centers as distributed file storage containers and decentralized software algorithms which can produce optimized layouts for graphs with millions of nodes and edges. But since the screen space limits the visualization on the client side, additional aggregation techniques have to be applied.

The presented approach uses the cluster-computing framework *Spark* to precompute several levels of detail as an abstraction to visualize a graph. Each level is stored in a distributed database, and the web client fetches data on the fly to render the interactive visualization. Node and edge aggregation are performed on the server to improve the client's performance In the first step, nodes are grouped based on their spatial distance. Afterward, edges are aggregated by removing links when their endpoints have been grouped. Alternatively, those edges can be visualized as loops to preserve the underlying graph structure. The output is a graph with weighted nodes and edges to indicate the number of elements they represent. Those two aggregation processes are performed iteratively for each level of detail and then divided into *tiles* for indexing. Tiles are rectangular raster or vector images used in digital mapping applications to separate large maps and to enable map navigation by seamlessly joining dozens of individually requested tile files.

The client fetches those tiles and renders them by using the *Fatum* library. Furthermore, the client applies a real-time edge bundling algorithm, based on a variation of the Kernel Density Estimation (KDE) method. By using an angular aggregation technique for long edges, visual clutter can be further reduced.



(a) (b) (c)

Figure 3.8: Visualization of the panama papers dataset. (a) shows an overview. (b) shows the details of a cluster. (c) shows a zoomed-in visualization of the cluster [PA18].

Figure 3.8 (a) shows an overview of the whole *panama papers* dataset. The graph has 839 thousand nodes with assigned labels and 1.2 million edges. The aggregation of nodes created a heat map showing the node density, enabling to spot clusters, which can be seen in figure 3.8 (b). Finally, the cause of the cluster can be seen in figure 3.8 (c).

Even though the current approach is only able to visualize static graphs, the underlying technology of distributed hardware and software has the potential to visualize large streamed graphs. One promising solution to achieve the rending of dynamic graphs is the so-called lambda architecture.

3.4 Discussion

When the size of data grows, the corresponding graph becomes cluttered and visually confusing, giving users more trouble discerning between nodes and edges [CQ07]. Two conventional approaches to address this problem can be found in recent literature. In *edge aggregation* methods, single edges are merged into bundles to reveal underlying patterns in the graph structure. *Node and edge aggregation* techniques cluster nodes by either mutual attribute properties or structural proximity. Additionally, edge aggregation is usually applied as well, since the number of edges often exceed the total number of nodes. By combining both aggregation techniques, groups of entities and the relationship between them can be revealed.

Methods working exclusively with edge aggregation have been far more popular in the early days of large graph visualization. On one side, this circumstance is due to the sheer amount of edges in a large graph. Compared to nodes, the number of edges in a graph is usually much higher. Furthermore, edges consume more space than nodes,

covering large parts of the visualization which ultimately leads to cluttered images. This problem is especially visible with edges spanning large distances by connecting nodes on opposing ends of the graph. On the other hand, many application areas and data sets are heavily focused on how nodes are connected and not actually on the individual nodes. For instance, visual analytics of raw trails data created through eye tracking is performed by analyzing the participant's optical flow visualized by succeeding edges. In this case, the individual nodes are needed for the graph structure, but their visual importance in the final visualization can be neglected. Therefore, the nodes are omitted, and only edges are rendered. Such algorithms have been created by Peysakhovich et al. [PV15] and Zwan et al. [VdZM16]. On the contrary, Weiwei et al. [WC08] and Hurter et al. [HC14] are rendering the individual nodes as decent spheres or circles.

It should be noted that dynamic edge bundling approaches based on some variation of *kernel-density estimation* calculations, such as proposed by Hurter et al. [HC14] or Zwan et al.[VdZM16], are useful for applications where the spacial edge position has either no relevance at all, illustrating only connections, or only indirect importance as in eye tracking applications. When spacial information has direct relevance, dynamic edge bundling techniques must be applied with care, especially for dynamic graphs. For example, when visualizing *distribution* changes of flight trails over days, dynamic bundling would only show the local spatial *mean* changes over time [HC14].

Node aggregation results in heatmap-like visualizations which enables the illustration of node clusters and structural characteristics. Additionally, edge aggregation techniques must still be applied to avoid visual clutter. Even though the abstraction of nodes adds another layer of information to the node aggregated graph, it also increases the complexity of the whole rendering task which has an impact on the overall performance. Such algorithms have been created by Zinsmaier et al [ZM12], Panagiotidis et al. [PA15] and Romat et al. [RH13].

Implementations on the GPU are more than a magnitude faster than a CPU implementation [BF14]. Performance is of utter importance when adding user interaction techniques to visualization or by using animated approaches. Delays caused by buffering or slow rendering processes can affect the user experience and obstruct the user's analyzation possibilities. Due to such necessities, methods relying on modern GPUs by exploiting all possibilities of the render pipeline, as shown by Zinsmaier et al. [ZM12], are by far more promising for applications where many tasks are performed on the client-side than conventional CPU based approaches.

By using the parallelism of cluster-computing frameworks like *hadoop* not only for layout algorithms but also for visualization methods, large graphs which are too big for the disk space of a single computer can be visualized. But this requires algorithms based on pre-computation steps since the server-side has to reduce the number of tasks normally done by the client. Perrot et al. [PA18] proposed such an approach, where the client

only renders pre-computed map tiles and handles user interaction. Even though the performance of computer clusters is tremendous, costs accumulate which makes such approaches currently much more expensive to sustain than traditional server-client methods.

The evaluation process of usability and user acceptability of the individual papers is a significant challenge, which has been applied pretty unilaterally by the selected papers. Only the method proposed by Peysakhovich et al. [PV15] has been tested and evaluated by actual user studies with participants and experts using eye tracking. On the other side, all presented approaches have been evaluated by measuring the runtime for one or more well-known data sets, each with a given node and edge density, and comparing the results with the overall performance of previous methods. Additionally, the general appealing of the final rendering has been analyzed by the authors. Although performance is indeed a significant factor for visualization algorithms, in my opinion, at least methods using color coding or opacity mapping techniques to illustrate additional attributes should have been tested on actual users. For instance, I was not able to comprehend the positive influence of the dynamically color-coded directions on bundled edges in Peysakhovich et al. [PV15] method or the advantage of visualizing textual attributes next to the individual nodes as used in Perrot et al.'s [PA18] approach.

The concept of a mental map and its impact in visual analytics is a topic vigorously discussed since the first publication of dynamic graph visualization [BF14]. Many papers have discussed the importance of preserving the user's mental map during aggregation and animation. For example, the layout process in the approach presented by Panagiotidis et al. [PA15] can be controlled interactively by the user to improve the stability of the mental map. While other approaches neglect the concept of a mental map entirely, stating that its contribution to the overall experience of the visualization has not been proven so far [BF14].

CHAPTER 4

Conceptual Overview

4.1 General Introduction

Graphs are a prominent data structure within Visual Analytics and related research fields to describe relationships between entities. These structures are described by a set of human-readable, ASCII-based file formats, where additional attributes can be attached to vertices and edges, to complement graph elements with application-related information. By creating visual representations out of these data collections, graph analysts aim to gain actionable insights and make data-driven decisions based on them [VLKS+11]. As graphs get larger, graph pre-processing is often needed for visualization to simplify complex data. Additionally, interactions with immediate feedback for the most frequent operations are added to help the viewer to solve tasks connected to graph exploration. These tasks can be of different natures, such as topology-based or attribute-based [LPP+06], where, for instance, connections between nodes or additional values of elements are analyzed. Furthermore, graphs can be subject to discrete changes, such as insertions or deletions of vertices or edges. The studying of these structural changes gives information on temporal processes.

This thesis proposes an interactive web-based visualization for large dynamic node-link diagrams. By using a combination of automatic graph layout generation and graph aggregation at multiple levels during the creation of a dynamic graph, complex graphs are aggregated and visualized as compound graphs, where nodes and edges are merged to single meta elements, revealing relationships between them. These level-of-detail constructs are created once during the creation process and can be fetched separately by the viewer. This allows users to change visual parameters, like the scale of a graph, at their convenience. Additionally, dynamic graph sequences are pre-processed gradually to identify added and removed elements between consecutive steps.

4.2 System Overview

The presented system consists of five individual components, as well as a database and a caching service, which are shown in Figure 4.1. This application has been designed as a server-client model. Hence, except for the web frontend, all components are executed on the server-side. The core functionality of each component is as follows:

- The **web frontend** is executed by the client and runs inside a common web browser. This allows the system to be cross-platform compatible without the need to install any additional software to use this application. After fetching data from the backend, the frontend's core functionality is the rendering of the graph visualization, as well as handling user input.
- The **front proxy** acts as a single entry point, default service gateway, and is the primary load balancer for customer requests. It translates the HTTP/1.1 calls produced by the client into HTTP/2 calls and forwards them to the backend service.
- The **backend service** processes all client calls and either delegates work further to the pre-processor, or fetches and edits data from both, the database and the cache. During the visualization of a graph, this is the only server-side component containing business logic involved in the process.
- When a new graph is created, the **pre-processor** handles parsing and transforming of the input file, the creation of a level-of-detail representation of the new graph, and the orchestration of the layout and sequence service to finish the overall processing.
- To provide a single system capable of both, calculating a graph layout and visualizing a graph, an existing **layout algorithm** published by Crnovrsanin et al. [CT15] has been included. This incremental force-directed layout method is used to create stable and aesthetic layouts of a graph.
- When processing dynamic graph data, the visualization of change is of utter importance. A graph sequence consists of individual graphs, so called *graph pages*, which represent the respective state of the sequence. After an individual graph has been pre-processed, the **sequence service** compares the current graph page with its predecessor and adds lifetime data to each graph element.
- A **database** is used to store the results of the pre-processing step and contains all data required for visualization.
- A centralized **cache** is used to share data between individual modules and to provide fast access to previously fetched data from the database.

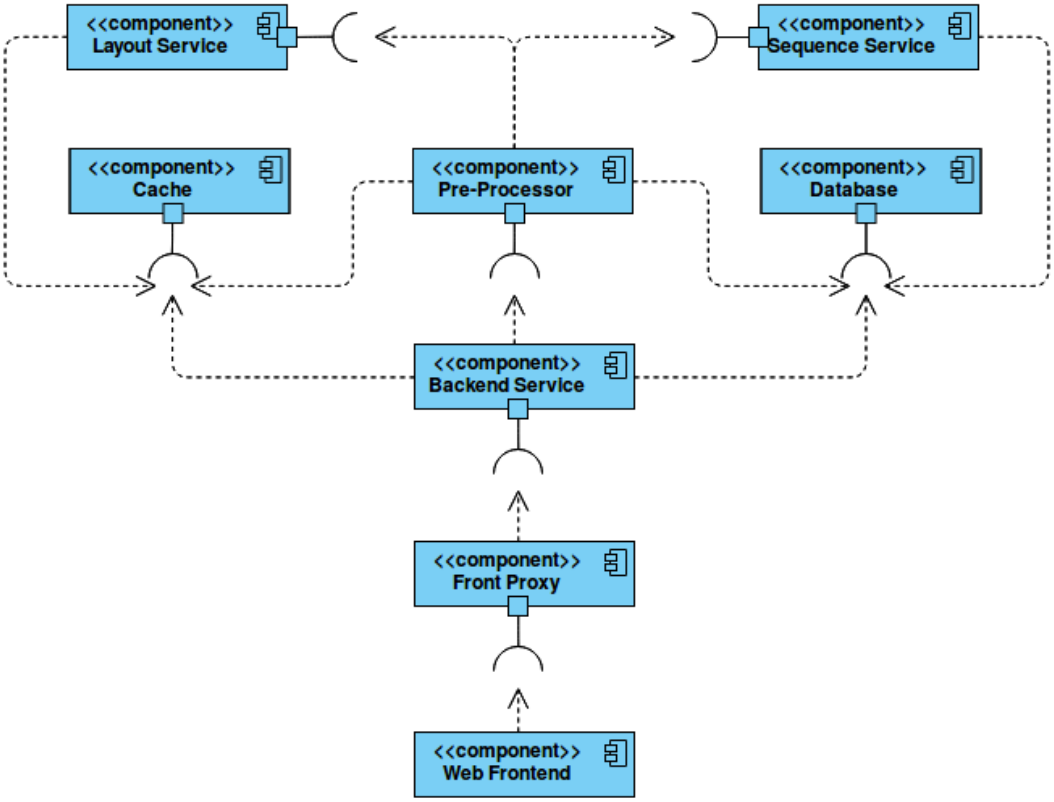


Figure 4.1: An illustration of the system's individual components, as well as the dependencies between them.

The presented system has been designed by using a component-based architecture to support the separation of concerns. Each component provides a set of functional services and exposes them through a provided interface [PV14]. Communication between components has been realized by exchanging direct procedure calls, blocking the sending process until the receiver has finished its work. This results in a process workflow, where all steps are processed sequentially. Data, on the other hand, is shared by using the caching service, which increases data retrieval performance while reducing the size of the exchanged messages between components.

The core functionality of this system can be divided into two tasks: Creation of a new graph and visualization of an existing graph. These two workflows are completely separated from each other. The components of the system, as well as their interactions, are described in detail by discussing the application's two main features.

4.3 Visualization of a graph

In this section, we discuss the used representation of graphs and implemented mechanisms for the user interface. Thus, navigation and exploration techniques, element selection, attribute visualization, hierarchy and aggregation, as well as the visualization of lifetime of elements, are discussed. Finally, the workflow to visualize a graph is described.

4.3.1 Visual representation

Beck et al. [BF14] and Landesberger et al. [VLKS+11] have identified the following two common visual representations of dynamic graphs:

- **Node-link diagrams**:
  The main challenge in using node-link diagrams is the creation of a readable layout so that certain notions of graph aesthetics are supported. Due to the design of node-link representations, a significant amount of empty space is used to separate individual elements and to provide a readable layout. Thus, scalability problems for larger or denser graphs can occur. [GFC04, VLKS+11].

- **Matrix diagrams**:
  Due to the non-overlapping display of graph edges and the readability of graphs, matrix-based representations are suitable to large or dense graphs. However, matrices suffer from scalability issues and increased difficulty for users to follow paths. Furthermore, many users are unfamiliar with matrices, which can have a negative impact on their readability [GFC04, VLKS+11].

Undirected node-link diagrams are used in this project to represent graphs since they are more intuitive to read and more commonly used than matrix diagrams [GFC04]. Therefore, a force-directed layout algorithm has been included in our approach to calculate a stable and aesthetic graph layout. Repeated graph aggregation is used to overcome the problem of scalability, creating a hierarchical compound graph. While the original data is shown on the lowest hierarchy level, meta elements represent nested sub-graphs at higher hierarchy levels. Nodes close to each other are merged into single meta nodes, while links between merged nodes are bundled and combined as well. The approach is used to overcome the overplotting problem by reducing the size of the graph and revealing relationships between groups of nodes [VLKS+11].

Change over time is visualized using an animated time-to-time mapping technique, where each part of the sequence is shown individually. The lifetime calculation considers both past and future time steps. Thus, information about newly created elements and elements that are going to be deleted in the next step are shown in each frame of the animation.

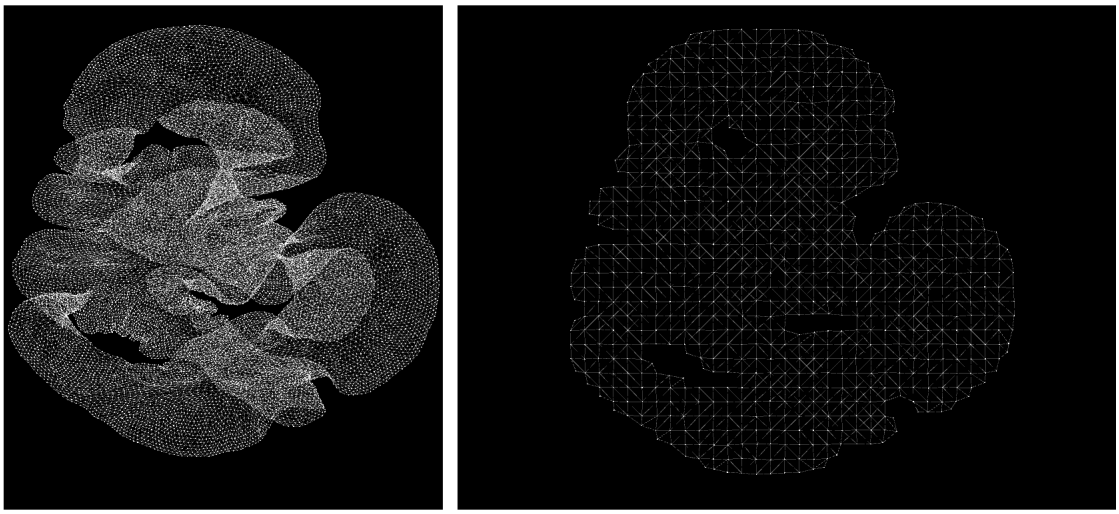


Figure 4.2: Main and detail view visualizing the 4elt data set. The detail view on the left shows the original data consisting of 15,606 nodes and 45,878 edges. An aggregated compound graph at the hierarchy level 5 of 14 is shown in the main view on the right, composed of 761 nodes and 4,482 edges.

The position of nodes is tried to be kept stable to enable the user to follow changes between consecutive parts of the sequence. Therefore, a layout algorithm that is able to ensure the stability of the layout and to preserve the *mental map* is used [CT15].

4.3.2 Navigation

Interactive visualizations helps users solving tasks connected to exploration of graphs, such as finding adjacent nodes, or determining connections between nodes [VLKS+11]. Therefore, the following two standard interaction techniques have been added to enable the user to navigate the graph interactively:

- **Brushing and linking**
  The visualization consists of two separate views, the main view and the detail view, allowing the viewer to compare data of different levels of abstraction. While the detail view always shows the original data of the graph, the main view shows the meta nodes at the selected level-of-detail. Figure 4.2 shows an example of the dual view approach. While the original data is shown on in the detail view on the left, a simplified version of the graph is shown in the main view on the right.

  The camera objects of both views, used to render each canvas, can be linked together to focus on arbitrary points in the visualization coherently. This enables the user to show the same regions in both views to compare them directly.

- **Panning and zooming**
  The viewer can navigate in both two-dimensional directions by pressing the left

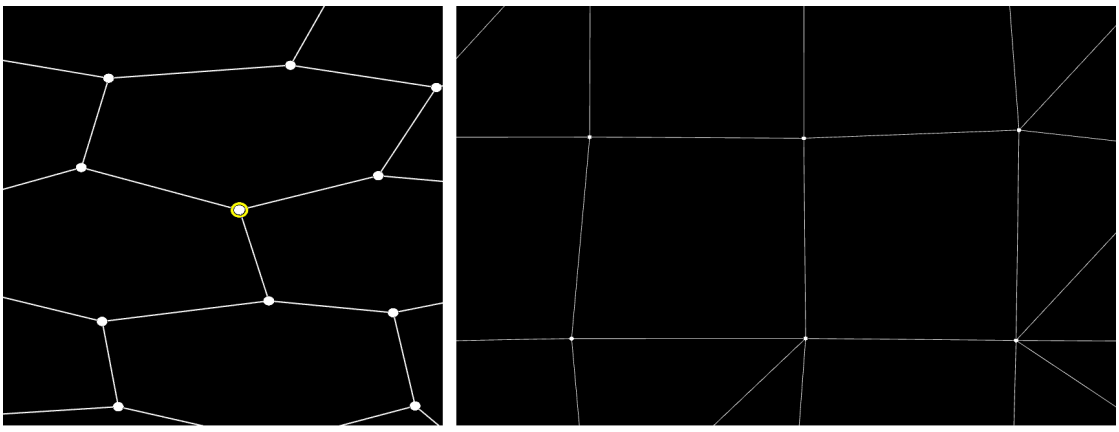
mouse button inside one of the HTML canvas elements and dragging the mouse in any direction. Additionally, the zoom-level in both views can be changed by rotating the mouse wheel inside the canvas element. The range of the zoom level is limited so that the graph elements never leave the area between the near and far clip plane of the perspective camera object. This approach prevents the graph from being clipped during the render process but can limit the application's ability to show extensive graphs as a whole on smaller screens.

Both functions do require to render both canvas elements once again, but none of them do not require costly operations. This allows immediate visual feedback without hindering the user with additional loading times while investigating the graph.

4.3.3 Selection and attributes

Each node of the visualization can be selected by the user, which is shown in the visualization with a yellow glow effect. Since the selection function is based on the position of elements and does not include information about hierarchical relations, the selection is not linked between both views.

Additionally, all domain-specific attributes associated with the selected node are listed in an HTML table element below the visualization. However, attributes are only stored for original data nodes and are not inherited by meta-nodes. Therefore, meta-nodes can be selected, but they do not provide attribute information.



Attribute	Value
label	100
x	8934.211426639435
y	8721.634136087225
w	5
h	5

Figure 4.3: A zoomed representation of the 516 data set, consisting of 516 nodes and 729 edges. A node is selected in the detail view. Five attributes of this node and their values are shown in the table below.

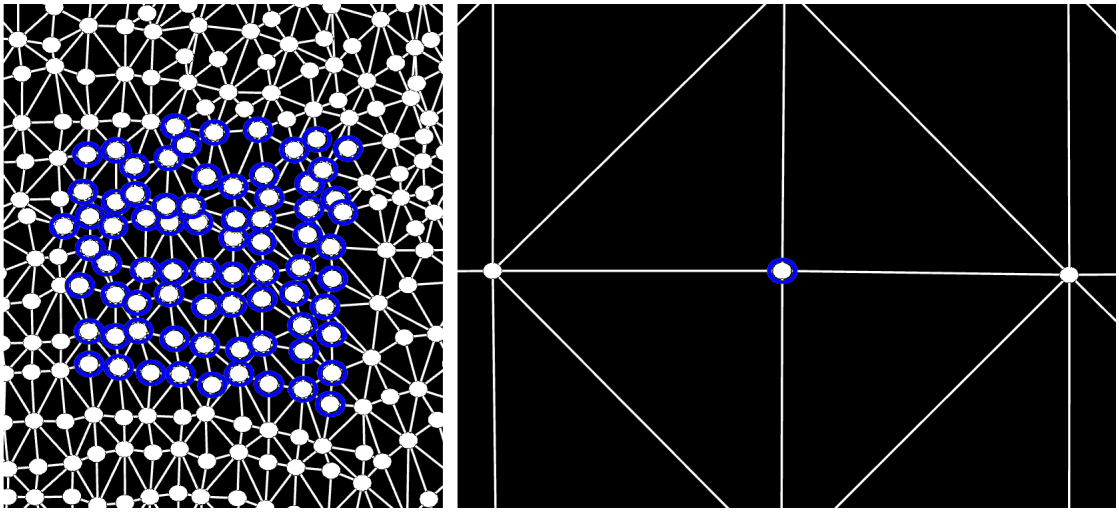


Figure 4.4: A zoomed representation of the Crack data set, consisting of 10,240 nodes and 30,380 edges, showing a top-down hierarchy visualization. The highlighted elements in the left view are aggregated by the meta-node element shown in the left view.

An example of node selection is shown in figure 4.3. A single node has been selected in the detail view, which is visualized by a yellow ring around the element. The node is linked to five attributes, which are shown in the table below. While the names of the attributes are listed in the first column, the respective values are shown in the second table column.

4.3.4 Hierarchy and aggregation

Graph aggregation creates a simpler abstracted version of the original graph data. This approach helps to overcome problems of clutter and showing the underlying structure of the graph. The relation between elements at different levels-of-detail is constructed with a hierarchy structure and visualized with a blue glow effect. This hierarchy can either be created in a top-down or bottom-up fashion. By clicking on a node element in the detail view, the corresponding meta-node in the main view is highlighted. On the other hand, all respective aggregated data nodes are highlighted when the user clicks on a node in the main view.

Figure 4.4 shows the hierarchical relationship between a single meta-node and multiple data nodes. All highlighted nodes in the detail view on the left are aggregated by the highlighted meta-node shown in the main view on the right.

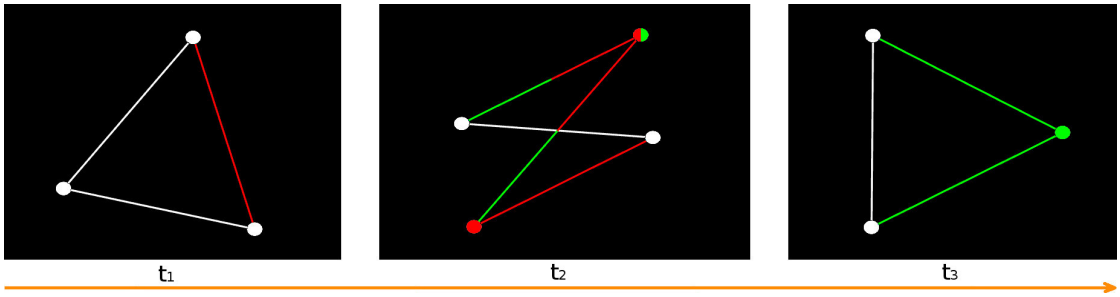


Figure 4.5: Illustrating example of a fully-dynamic graph sequence on a timeline with element creation and deletion over three time steps.

4.3.5 Lifetime of elements

The evolution of a dynamic graph is visualized by highlighting nodes and edges which have been subject to discrete changes over time. Therefore, each element's state corresponds to a particular part of the dynamic sequence, and is either *unchanged*, *added* or *deleted*. While added elements are highlighted when they first appear in the visualization, deleted objects are marked one timestep before their actual deletion.

Figure 4.5 shows a visualization of a dynamic graph sequence with a length of three frames. In the first frame, the graph with its elements is created. Usually, created elements are rendered in green color, but to avoid the overuse of color in the visualization, elements in the first frame are shown in white instead. The red edge at t_1 indicates that this element is going to be removed in the next frame. At t_2, one node and two edges are created and immediately removed at t_3. Thus, two colors are used during the rendering of their texture, resulting in a red-green-striped appearance. Furthermore, one node and one edge are marked for deletion at t_2. Finally, a single node and two edges are created at t_3.

4.3.6 Workflow

The conceptual overview of the visualization of an existing graph is shown in figure 4.6. First, the frontend fetches all available graph sequences from the backend. After the user has selected a graph sequence for visualization, the frontend requests the node and edge data from the first graph of the sequence. Since the visualization contains an overview and a detail view, both, the highest and lowest level-of-detail, are fetched initially. Additionally, the length of the graph sequence and the number of detail levels are fetched from the backend. This allows the user to select a specific level-of-detail in the overview interactively, or to visualize another part of the graph sequence.

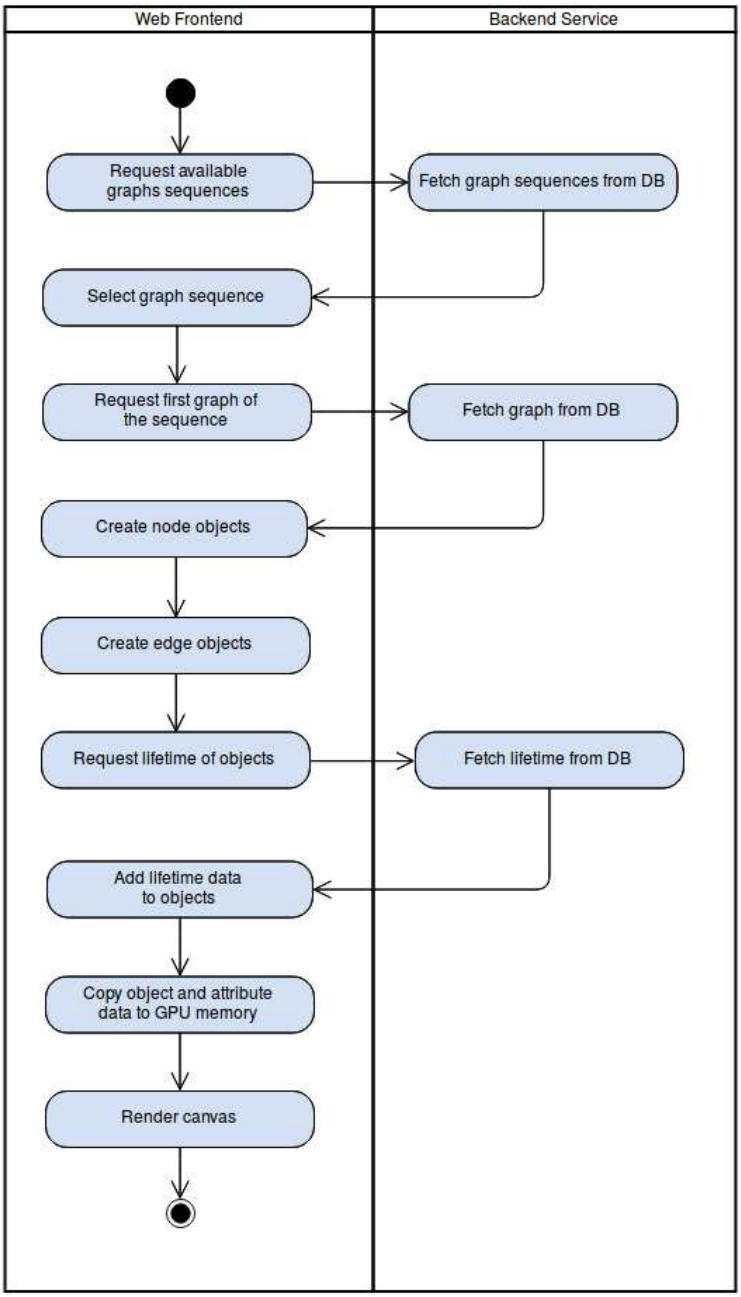


Figure 4.6: A conceptual overview of the workflow of the sequence service.

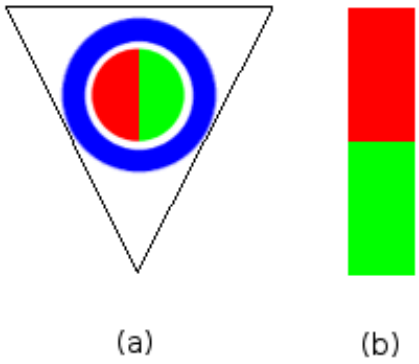


Figure 4.7: The textures used to render node and edge elements. (a) shows the triangle texture of node. The blue ring represents the highlight area of a selection. (b) shows the rectangular texture of edges.

Nodes are represented in data as a point cloud and edges contain information of the respective start and end nodes. To be able to render nodes and edges as two-dimensional objects, the data fetched from the backend must be processed first. Each node is replaced with three points forming a triangle with its center positioned at the place of the node. For each edge, four points forming a rectangle going from the start node's position to the end node's position, are created. This approach allows the frontend to be in full control over the size of the visualized objects.

Additionally to the positional data, a flag for selected objects and for the hierarchical visualization is stored to each node element. This allows the user to highlight individual objects after the visualization has been created.

Figure 4.7 shows the textures used for the rendered elements. The actual form of nodes is defined in a partially transparent texture in the shape of a triangle. 4.7 (a) shows the round shape used to visualize node elements. The blue ring is used to add a highlight effect to selected nodes and to show hierarchical relations. If the respective data flag for this element is not set, this part of the texture is ignored during rendering. 4.7 (b) shows the rectangular texture used for edges. In both textures, different colors are used to determine specific regions of the texture in the fragment shader. The actual color of the visualized objects is set separately. Red and Green areas represent the element itself and their rendered color depends on the object's lifetime. When the element has either been created or will be deleted in the next step of the sequence, a single color is used to visualize both areas. However, when the element has been created and will be deleted in the next step, different colors are used for the areas. Therefore, two lifetime states can be visualized at the same time.

After the objects have been created, the frontend fetches the lifetime data from the backend. Since the default lifetime is ”unchanged”, only the IDs of created and deleted elements are transmitted. When the lifetime data has been updated, all object information is copied to the GPU memory and WebGL renders the canvas.

4.4 The creation process of a new graph

In this section, the process of creating a new graph is discussed. First, a conceptual overview of the workflow is given, describing the involved components and their respective purpose. At last, the two most essential components of the creation process, the pre-processor and the sequence service, are discussed in detail.

4.4.1 Workflow

A conceptual overview of the process to create a new graph is shown in figure 4.8. First, the user opens the web frontend to either select an already existing graph sequence or to create a new one. Afterward, a data file containing information on a static graph is uploaded to the server. The backend collects the file content and additionally creates metadata to store information about the file and the relation to the selected or created dynamic graph. The pre-processor parses the file content and converts it into a domain-specific format. This transformation ensures that the layout service can process the file correctly. After the layout service has created the graph layout, the pre-processor creates a level-of-detail structure. To visualize the change in dynamic graphs, the sequence service compares the newly created static graph with its predecessor in the graph sequence and adds lifetime attributes to each element.

Figure 4.9 shows the interactions between the individual components during the creation of a new graph. It can be seen that the communication between the components resembles work requests, where the following tasks are delegated to other components. After a processing step has been completed, a response containing the respective result status, either true or false, is sent back to the caller.

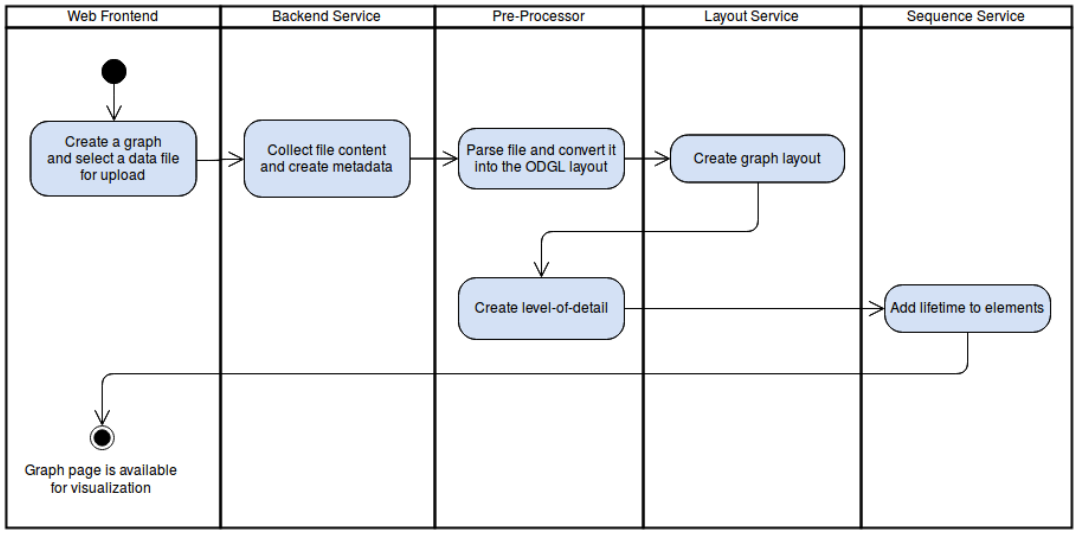


Figure 4.8: A conceptual of the workflow to create a new graph page.

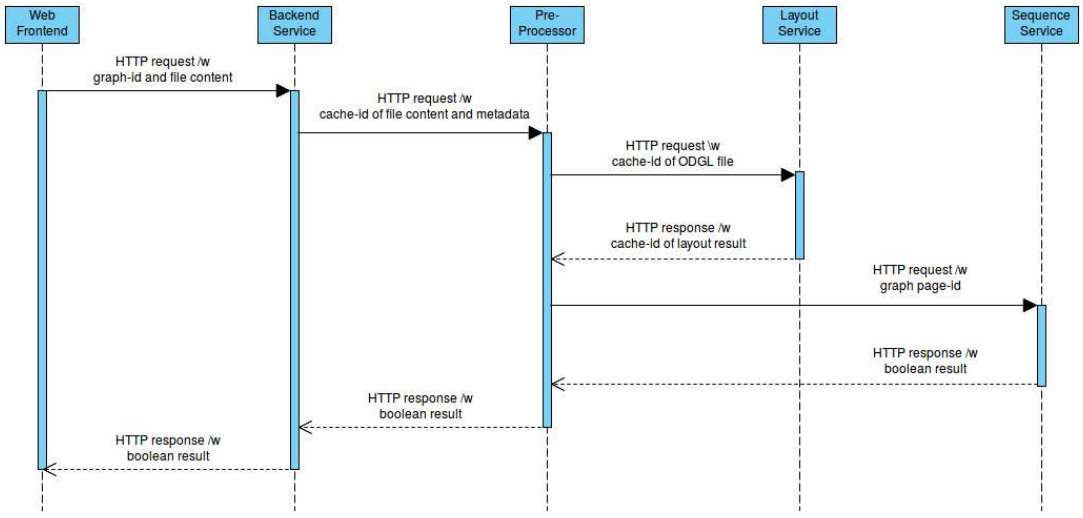


Figure 4.9: The interactions between the individual components during the creation of a new graph page.

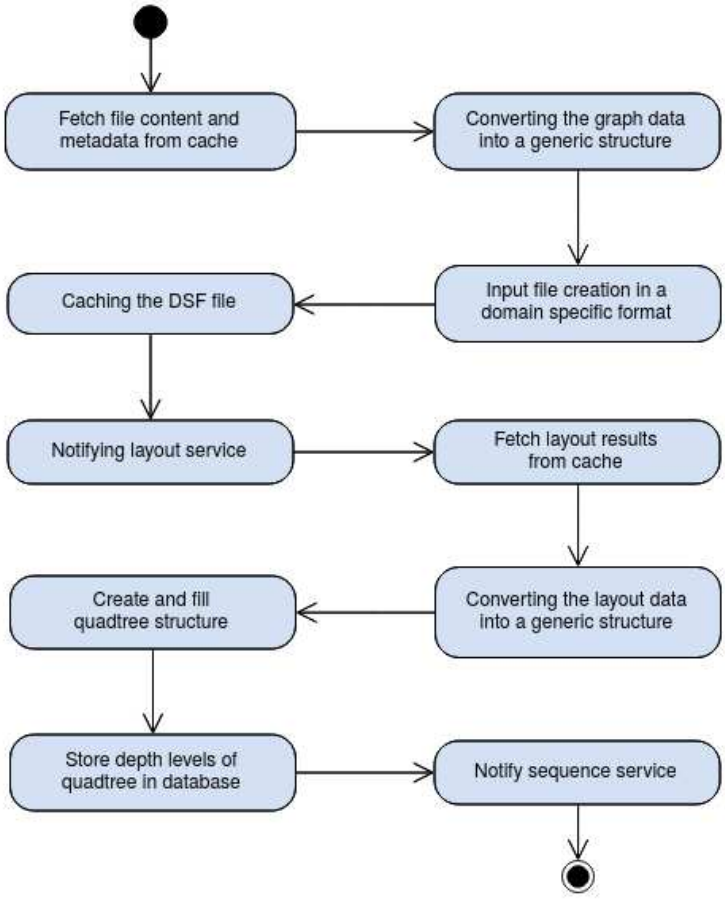


Figure 4.10: A conceptual of the workflow of the pre-processor service.

4.4.2 Pre-Processor

Figure 4.10 shows a step-by-step description of the pre-processor's workflow. The pre-processor service supports three common file types: Comma Separated Values (CSV), Graph Modeling Language (GML), and Matrix Market (MM). Different data file types support a diverse set of information. For instance, while the CSV format contains data about individual nodes and their relations, the GML format additionally provides a node label and multiple attribute descriptions. Therefore, all file types are parsed and transformed into a generic representation of the graph. Each generic node consists of a unique ID and optional attribute values, while generic edges store the IDs of the respective start and end node. At this point, the data is relational and does not provide positional information. To obtain node positions, the layout service must be notified to calculate a two-dimensional layout of the graph. Since the layout service uses a domain-specific format, the generic nodes must be transformed and written into a text

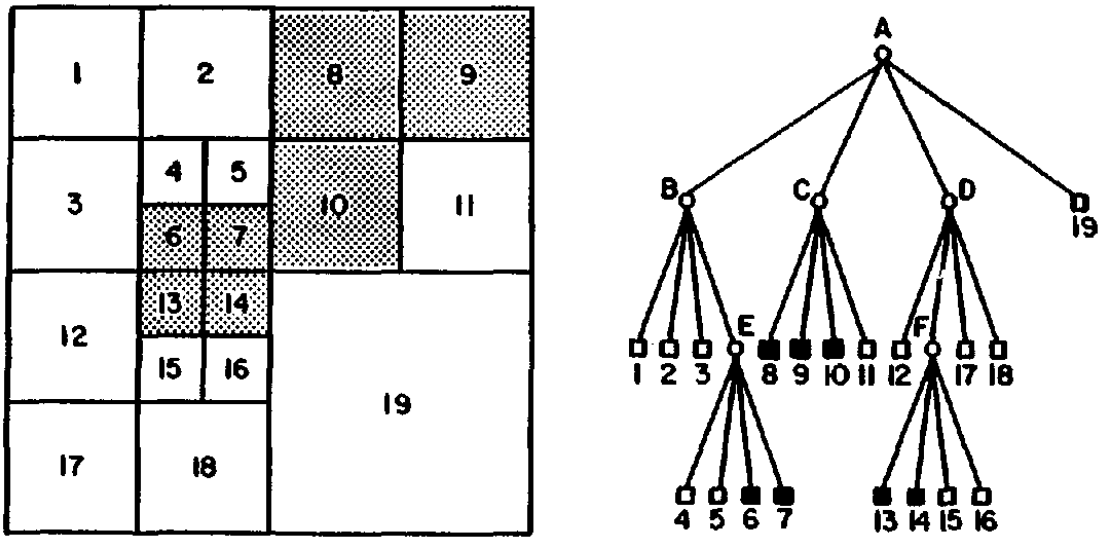


Figure 4.11: Spacial decomposition and its quadtree [Sam84]

file first, which is forwarded afterward to the layout algorithm. After the layout results have been fetched, the containing node data is transformed back into their generic representation, but complemented by their respective two-dimensional position. Next, a *quadtree* data structure is used to create multiple detail levels of the graph. Afterward, each level-of-detail layer is stored in the database, and the sequence service is notified.

Quadtree data structure

The concept of using a quadtree data structure for hierarchical aggregated visualizations has been introduced by Elmqvist et al. [EF09]. In the workflow of the pre-processor service, the quadtree structure is created and filled with the generic node and edge data to construct a level-of-detail structure. Each cell of the quadtree spans a specific area in the form of a rectangle and can be linked to precisely four child cells, whereby each child cell is positioned in one-quarter of the parent's spacial area. Therefore, two-dimensional space is recursively subdivided it into four quadrants. While the actual positional data is stored in leaf cells, parental cells are positioned at the average position of its child cells. This approach results in a data structure of a specific depth, where each depth level represents an abstraction of the positional information stored in lower levels. Therefore, each layer of the quadtree can be equated with a level-of-detail representation of the graph.

An example of this space-filling technique is shown in figure 4.11. Highlighted cells contain the data which has been added to the quadtree structure. Node A, at the top level, is called *root* and encompasses all spacial areas. This space is decomposed into four equal quadrants with each subsequent level. The nodes $A - F$ have been split into sub-quadrants and have exactly four children. The remaining nodes $1 - 19$ have not been

divided any further. They are called *leaf nodes* and are either empty or hold part of the inserted data.

Algorithm 4.1: The algorithm to add a generic node to a quadtree cell.

```
1:  procedure ADD_NODE(x, y, node_id)
2:      index ⇐ get_sub_area_index(x, y)
3:      if self.is_leaf then
4:          if self.x is x and self.y is y then
5:              return error
6:          end
7:          self.is_leaf ⇐ false
8:      end
9:      if self.children[index] exists then
10:         id ⇐ self.children[index].add(x, y, node_id)
11:     else
12:         id ⇐ create_new_child(index, x, y, self.id)
13:     end
14:     self.calc_avg_pos()
15:     return id
```

The total size of the graph layout is calculated first by iterating through all nodes. Afterward, the quadtree's root cell is created at the center of this area. The list of generic nodes is processed iteratively, and each node is added to the quadtree's root cell.
The procedure to add a new node to the quadtree is shown in algorithm 4.1. First, the cell calculates the respective sub-section of its spacial area with the node's position. This process is done by the function *get_sub_area_index*. Next, leaf cells are processed. If there is already a leaf cell containing the added node position, an error is returned, and the duplicated node position is not added to the structure. Afterward, the data is added to a child node. Therefore, the leaf flag of the cell is removed. If no child cell has already been created for this sub-section, a new cell is instantiated at the position of the node and assigned to the respective area. However, if a child cell already exists, the add node method is called on this cell instead. After the recursive call to find a leaf cell for the given node has been completed, the parent node recalculates its position by taking the average position of its child cells.

A distinct feature of the quadtree is the generation of unique IDs that are added to each cell. By encoding relational information of the graph structure into each cell ID, it is possible to extract the exact depth of each element, as well as child-parent relations of cells. While the root element is assigned with $id = 0$, the identifiers of child cells are created recursively when they are added to the structure. The index of the child element is appended for each depth level of the graph to the cell ID. An example of a small graph with the relational ID structure is shown in figure 4.12. The first depth level consists of four meta-notes, each with an ID similar to their child-index of the root element. The second child has been divided twice and therefore contains two child nodes - one at the

index two and the second at the index three. Their IDs have been created by appending their respective indexes to the ID of their parent element.

After all generic nodes have been added to the quadtree, the collection of generic edges is processed iteratively as well. For each element, the quadtree searches for the leaf cells containing the start and end position of the edge. Afterward, the tuple containing both cell IDs is added to the quadtree's edge list.

Algorithm 4.2: The algorithm to get all quadtree data cells of a specified depth level.

```
1:  procedure COLLECT_NODES_AT_DEPTH(depth, collection)
2:      cell_depth ⇐ extract_depth_from_id(self.id)
3:      if cell_depth > depth then
4:          return
5:      end
6:      if self.is_leaf or cell_depth is depth then
7:          collection.add(self)
8:      else
9:          for child in self.children
10:             child.collect_nodes_at_depth(depth, collection)
11:         end
12:     end
```

In the next step of the pre-processor workflow, the data contained by the quadtree is extracted per depth-level and stored in the database. Since not all branches of the quadtree structure are of similar depth, leaf cells in higher levels must be included as well. For instance, the complete set of data cells of the second depth level in the graph shown in figure 4.12 consists of the IDs 1, 3, 22, 23, 42, 43 and 44.

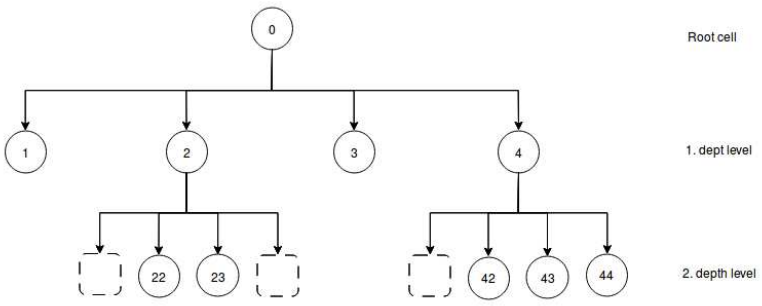


Figure 4.12: A small graph showing concept of relational cell IDs. The root cell is given the $id = 0$. Child elements have the path from the the root cell encoded.

The procedure to extract all cells for a specific quadtree depth level is shown in algorithm 4.2. First, the depth of the current cell is calculated. If the depth of the current cell exceeds the given depth, the remaining cells of the branch are not further processed. Next, either leaf cells and meta cells at the specified depth are added, or the element's child cells are processed instead.

The process of extracting edges of a given depth works similarly. The depth of the start and end nodes are clipped to the given depth level. Afterward, duplicated edges are removed to reduce the total amount of data.

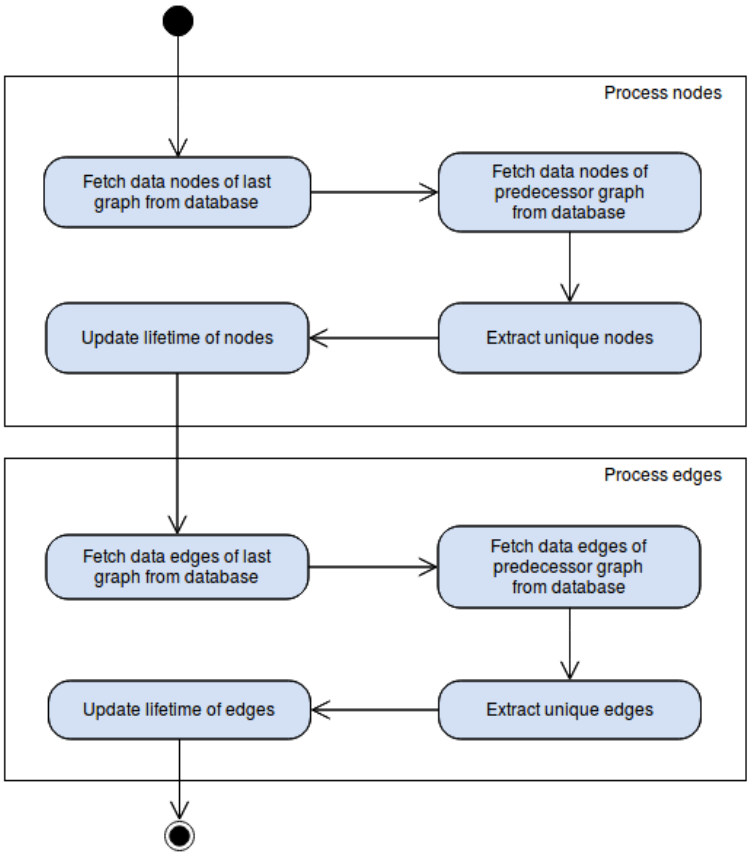


Figure 4.13: The workflow of the sequence service to add lifetime information to elements of a graph sequence.

4.4.3 Sequence Service

Figure 4.13 shows the workflow of the sequence service, which consists of two similar parts. First, all leaf nodes of the last graph and the previous graph of the dynamic sequence are fetched from the database. Afterward, both sets are compared to each other using the quadtree cell-id of each element created by the pre-processor service. If an element with a specific cell-id exists in both sets, it is removed from the collections. Thus, after all elements have been processed, both sets contain a collection of unique elements. The change occurred in the last step of the graph sequence is split up in those collections. The first set, which initially contained all elements fetched from the previous graph, contains all deleted items, while the second set, which initially included all elements fetched from the last graph, holds all newly created elements.

Algorithm 4.3: The procedure to compare the nodes of the last and the previous graph.

```
1:  procedure EXTRACT_UNIQUE_NODES(collection_a, collection_b)
2:      collection_a.sort()
3:      collection_b.sort()
4:      diff_a ⇐ new collection
5:      diff_b ⇐ new collection
6:      while collection_a is not empty or collection_b is not empty do
7:        if collection_a is not empty and collection_b is not empty then
8:          last_a ⇐ get last element of collection_a
9:          last_b ⇐ get last element of collection_b
10:         if last_a is equal to last_b then
11:           remove last elements of collection_a and collection_b
12:         else if last_a > last_b
13:           move last element of collection_a to diff_a
14:         else
15:           move last element of collection_b to diff_b
16:         end
17:       else if collection_a is not empty then
18:         move last element of collection_a to diff_a
19:       else
20:         move last element of collection_b to diff_b
21:       end
22:     end while
23:     return diff_a and diff_b
```

The procedure to extract all unique nodes of two sets of elements is shown in algorithm 4.3. First, both collections are sorted in descending order. This allows removing identical elements from both lists in a single iteration of the while loop shown in line 6. By comparing the last elements of the sets and moving unique nodes to separate collections,

the initial sets are reduced iteratively until both have been empty. When one collection is empty, no further comparisons have to be made, and the elements can be moved directly to the respective difference collection.

The procedure to extract unique edges works similarly. However, edges do not contain a quadtree cell ID. Thus, a representational ID is calculated for all edges first, by merging the cell-ids of the respective starting and ending node. Afterward, the previously described algorithm can be applied to extract unique edges.

After every unique element has been extracted, their lifetime values are stored in the database. The default value for lifetimes is 0. For deleted nodes and edges in the previous graph, the lifetime is increased by 2, while for newly created elements in the last graph the lifetime is increased by 1. This approach allows to store multiple lifetime states. When an element is created in a graph and deleted in the next part of the sequence, its lifetime value is 3.
Whenever a new graph of a dynamic sequence is added, the lifetime of newly created elements is calculated at the same time. On the other hand, the lifetime of deleted elements is only added when the next step of the sequence is added and processed.

CHAPTER 5

Implementation Details

This chapter covers details about the implementation of our prototype called DaGO (Dynamic Large graph Observer). An overview of the most important used technologies, frameworks, and libraries is given. Furthermore, implementation details, technical implications, and limitations are discussed. The main features of the UI are listed and explained in detail. The chapter concludes with the final prototype used to evaluate our system, shown in figure 5.3.

5.1 WebGL and structural implications

WebGL is a cross-platform, royalty-free web standard for a low-level 3D graphics API based on OpenGL ES. JavaScript can use the library for rendering interactive 2D and 3D graphics in a standard web browser without the need for additional plug-ins [Khr19b]. WebGL 1.0, which has been used in this project, exposes the OpenGL ES 2.0 feature set and has been extended by two individual entries of the Khronos WebGL extension registry [Khr19a]. First, the *OES_vertex_array_object* extension is enabled to provide vertex array objects, which encapsulate vertex array states and offer a convenient way to communicate draw command setup states to WebGL. Second, the *OES_element_index_uint* extension is enabled to allow visualization of larger graphs. By default, WebGL 1.0 uses a counter-argument of type *unsigned short* to specify the total number of vertices to draw. This two-byte variable has a maximum value of 65 535. The extension adds support for a counter-argument of type *unsigned integer*, which has a maximum value of 4 294 967 295, allowing rendering of larger structures.

Current state-of-the-art visualization techniques, able to render large graphs, are using advanced general-purpose GPU programming algorithms and make use of advanced shader stages [ZM12, LZ13, PV15]. Multiple render passes, in combination with the

geometry shader stage, allows merging elements and creating primitives on the graphics card. This approach has the advantage that after the graph data has been copied into the GPU's memory once, all further processing steps can be applied directly in the OpenGL pipeline, which can reduce the need for additional storage and improve the overall performance of the application.

Since WebGL only supports the *vertex shader* and *fragment shader* stages of the render pipeline, all processing steps limited by the reduced extent of render stages are moved to the server-side instead. The creation of a level-of-detail structure in the preprocessing step enables the frontend to fetch a specific set of data from the backend. Thus only a limited number of primitives have to be created before the graph can be rendered. This reduces the amount of memory storage in the browser and improves the performance of the application.

5.2 Communication

This application has been designed as a microservice-based architecture, where individual components are encapsulated in separate microservices. Each service instance of the distributed system runs as an isolated process on the host system. Therefore, we are using gRPC (Google RPC) [Goo19], a cross-platform and language-independent RPC system, as an inter-process communication protocol for service interactions. gRPC uses *Protocol Buffers* as the interface definition language for describing both, the service interface and the binary serialization of the message payload. The interface definition files are processed by the protocol buffer compiler to generate data access and service definition classes in various programming languages. Thus, adding support for polyglot implementation, where the server and the client can be written in different programming languages, as shown in figure 5.1. These source files are included in the respective microservices and include the following features:

- On the server side, a gRPC server is started to handle client requests by decoding incoming requests, executing service methods, and encoding service responds.
- On the client side, a local object called *stub*, which implements and calls the remote service methods can be created.
- Message wrapper objects provide accessors for each message field and methods to populate, serialize, and retrieve protocol buffer messages.

Protocol buffer messages are encoded in binary format, which is not human-readable, but smaller in size than text-based message formats like XML or JSON. Kiraly et al. [KSKB17] compared various RPC methods and concluded that protocol buffers are faster to serialize than XML based RPC methods and that they create messages of smaller size, which results in higher overall performance.

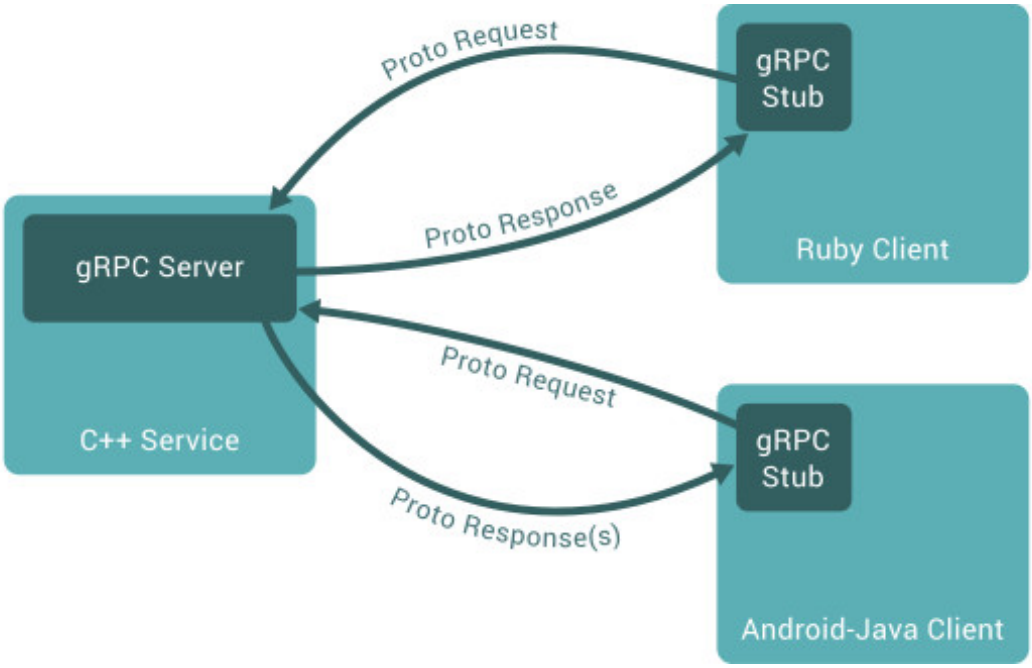


Figure 5.1: Architecture diagram of gRPC [Goo19].

gRPC uses HTTP/2 as the underlying multiplexed transport protocol to allow unary and server-side streaming. However, the current JavaScript library *gRPC-web* does not support HTTP/2 communication. Therefore, the Envoy service [Env19] is used in our infrastructure as a *front proxy*, which acts as an API-gateway to provide communication between the clients and the backend.
Envoy supports all of the HTTP/2 features required for gRPC requests and responses, translating incoming HTTP/1.1 calls from the web browser into HTTP/2 calls that can be handled by the backend service, and vice versa. Thus, the two systems are complementary.

5.3 WebAssembly

Hypertext Markup Language (HTML), Cascading Style Sheets (CSS) and JavaScript (JS) are the core technologies for building web pages, and they are used in the web frontend of this project. HTML is used to define the basic structure of the web page. CSS describes the presentation, formatting and layout of web sites and it's individual elements. JS is used to control the behavior of different elements, to process user input, to communicate with the backend service, and to control the user flow of the application. Although JS would be able to process the more complex tasks of the frontend service like data handling, objects creation, and setup and control of WebGL, these tasks have been moved to a fourth component, WebAssembly (Wasm).

Wasm is a binary instruction format for a stack-based virtual machine, which can be run safely in modern web browsers. It is a low-level assembly-like language that runs with near-native performance in a memory-safe, sandboxed execution environment. Furthermore, Wasm is designed as a portable target for high-level programming languages like C++ or Rust. This gives programmers low-level control and reliable performance [HRS+17b, W3C19].

Since gRPC does not provide any client for WebAssembly, the communication between frontend and backend has been implemented in JavaScript. After the positional graph data has been fetched from the backend, node and edge values are stored in separate *Float32Array* structures. Both arrays are then passed to the Wasm module, and object data is created, which can be used by WebGL. While triangles represent nodes, edges are shown as rectangular objects. Either way, *vertices*, *indices* and *texture coordinates* are created for each element and stored in individual dynamic array structures. Additionally, information about the *lifetime* of an object and a *flag* for hierarchy visualization and node selection is stores in separate arrays and initialized with a default value. Furthermore, the *database ID* of nodes and edges is stored as well.

After the object data has been created, JavaScript calls the render function of the Wasm module to generate the visualization of the graph in the HTML canvas element. First, *WebGL buffer* objects, which hold the object data, are initialized and collected in *vertex array objects*. Afterward, the shader programs are compiled and bound together with the vertex array objects, and the rendering is started. Consecutive render calls, which are bound to user navigation of the HTML canvas object, reuse already existing WebGL objects.

Lifetime data is fetched and processed separately. The backend service sends only the ids of either created or deleted node and edge elements. For each ID, the lifetime value is stored to the respective object, and the render function is triggered. Thus, changes in the lifetime data are applied to the visualization.

5.4 Hierarchy visualization and node selection

Relational information of meta-nodes and the respective aggregated node elements is stored in the so-called quadtree ID, a 64-bit unsigned integer, where each digit is in the interval of $[1, 4]$. These unique IDs are generated during the creation of the quadtree data structure that is used to generate a level-of-detail representation of the graph, which is described in chapter 4 - Pre-Processor. This results in a level-of-detail pyramid, where the complete graph data is located at the bottom, and the most aggregated representation is situated at the top.

Hierarchy visualization shows the structural aggregation of nodes across multiple level-of-details and can be created in two different ways. On the one hand, the hierarchy can be

constructed bottom-up, where proceeding from a selected node element, the respective meta-nodes in higher level-of-details are visualized. This approach is the fastest way to construct a hierarchy since all required relational information is already encoded in the quadtree ID of the starting node. By destructing the quadtree ID, removing one digit at a time, the quadtree IDs of all meta-nodes can be found. For instance, the selected quadtree ID 4241 is dissembled into 424, 42 and 4. These three meta-nodes are aggregating the starting node in this example.

On the other hand, the hierarchy can also be constructed top-down. This way, all aggregated nodes of a single meta-node are shown. Depending on the level-of-detail of the starting meta-node, this approach can be computational expensive and time consuming. First, the quadtree IDs of all possible child nodes in deeper levels of detail must be found. Since there is no information about which child nodes are actually existing, the list of possible nodes can be comprehensive. For instance, the meta-node with the quadtree ID 43 in a level-of-detail pyramid of size 10 has $87,380$ possible child nodes.

Either way, the backend service generates a list of quadtree IDs for a respective starting node and hierarchy type and fetches all existing nodes with a quadtree ID that is contained in the list from the database. Afterward, a list of the fetched node IDs is returned to the web frontend to visualize the requested hierarchy.

The web frontend offers the possibility to either selected a node or to use it as the starting point for hierarchy visualization by clicking on the elements with the right mouse button. The clicked node is identified by transforming the mouse coordinates in the canvas element into world space and applying the nearest neighbor search. Afterward, the unique entity ID of the node closest to the calculated point is returned to the frontend. The mechanisms to select nodes and to visualize hierarchy work similarly. While the returned node ID is passed directly to the selection procedure, the ID is first sent to the backend when visualizing hierarchy. In the latter case, the returned list of node IDs is passed to the selection algorithm instead.

Each node consists of an additional flag value, which stores the state of the element. The differentiation between multiple states works similar to the UNIX permission system. Figure 4.7 (a) shows the texture used by the shader to render nodes. While the core part of the image is used to visualize the actual node, the blue ring around it is used to show the hierarchy and selected nodes. The color and visibility of this ring depend on the value of each node's flag value. The following scenarios are possible:

- 0: Default value.
  The transparency of the ring is set to the minimum to hide the element.

- 1: Node is selected.
  Transparency is set to the maximum, and the ring is rendered with the color yellow.

- 2: Node is marked for hierarchy visualization.
  Transparency is set to the maximum, and the ring is rendered with the color blue.

- 3: Node is marked for hierarchy visualization and is selected.
  In our application, the selection state overrides other states. Therefore, the ring is shown as selected.

5.5 Database and caching

This project uses PostgreSQL 9.5, an open-source object-relational database system. All data required to create a visualization of a graph sequence, which has been uploaded and processed, is persisted in this database. Therefore, each successfully processed graph can be reused by multiple users.

Additionally, Redis 5, an open-source in-memory data structure, is used as a cache. The caching service is used for the following tasks:

- When file content is shared between micro-services, the file data is persisted in the cache, and only the created cache-ID is added to the payload of the gRPC message to reduce the message size of inter-process communication.

- The size of files describing graph data can be of multiple megabytes. Thus, a *chunked file upload* process is used when uploading a graph file from the web frontend. The original file content is split into multiple chunks with a maximum size of 512 kibibytes. Each chunk is uploaded separately to the backend service and stored in the cache. When all chunks have been uploaded successfully, the backend service fetches all individual chunks from the cache and merges them back together, before initializing the pre-processing. This prevents to exceed the maximum size of a gRPC message, while still being able to upload large files.

- The backend service caches the payload of response messages to improve the performance of similar consecutive frontend requests.

5.6 Programming languages and frameworks

Rust is a system-level programming language for developing reliable and efficient systems. It's rich type system and ownership model guarantees pure Rust programs to be free of memory errors and data races. Fine-grained control over memory representation and the absence of a runtime or garbage collector makes Rust programs fast and memory-efficient [MKI14].

Both, the pre-processor and the sequence service have been implemented in Rust 1.37. Furthermore, the WebAssembly module of the web frontend is written in Rust 1.37 +nightly and compiled into a Wasm library with a 32-bit address space.

Java is a strictly object-oriented, high-level programming language, designed for developing platform-independent applications. *Spring* is a modular framework that complements Java by providing an inversion of control container, and features like a web framework, and an abstraction layer for transaction management [Hem06]. The backend service is implemented in Java 8 and uses the Spring 2 framework.

C++ is a cross-platform, object-oriented programming language that extends the C language. By enabling a high level of control over system resources and memory, it can be used to create high-performance applications [Str00].

The standalone version [Tar19] of the layout algorithm presented by Crnovrsanin et al. [CT15] has been included in this project to generate positional data of nodes in a graph. The codebase of the layout algorithm is implemented in C++ 11 and has been extended by a gRPC server and a Redis client.

5.7 Containerization and scalability

Container technology refers to a software entity that can be regarded as a wrapper around a specific process, which runs in user space on top of an operating system's kernel. Unlike hypervisor virtualization, where multiple machines run virtually on physical hardware using a virtual-machine-monitor layer, the shared host kernel of containers is considered a lean technology, where only limited process overhead is required [Tur14]. But both technologies offer the possibility of separating individual processes and environments from the host.

This project has been designed with reproducibility in mind. Therefore, one goal of this project is to reduce the required effort to set up and run this application. Thus, the open-source engine for containerization, Docker [Doc19], has been used to achieve simplicity and parity. By moving all required components and dependencies into a specific environment handled by Docker, it is ensured that the system is running consistently on different computer systems. Since the Docker system is capable of downloading, building, and managing the specified environments automatically, the process of orchestrating Docker to set the application up and to execute it, can be automated via shell scripts.

The following list gives an overview of the most critical factors which have been encapsulated in docker environments:

- Multiple programming languages have been used to implement this software solution, which requires different build and runtime environments to execute the individual components.

- Different build automation system is used to manage dependencies and to compile, link, and package the code into an executable form.
- The communication between micro-services has been realized via a combination of gRPC and Protocol Buffers. Since the protocol buffers definitions files are language-neutral and platform-neutral, a specific compiler, code generator, and a gRPC compiler are required to create and include all software components required for communication. Furthermore, the code-generator, as well as the gRPC compiler, are language-dependent, which means that the respective software must be installed for each used programming language.

Another advantage of using containerization is flexible horizontal scalability. Docker automatically creates a separate network interface and connects the individual containers to it, whereas only specific ports of specific systems are accessible from the outside. Furthermore, individual containers can be multiplied on-demand and added to the network. By default, Docker uses DNS Round Robin for the load distribution of redundant service hosts. Additionally, the Envoy Front-Proxy is used as a single point of entry to the network, as well as a load balancer for the backend micros service. This gateway gives fine-grained control about the session and connection handling between clients and the backend service.

5.8 DaGO - control elements

Figure 5.3 shows the interface of our DaGO prototype. It contains the visualization of the dynamic data set reptilia tortoise network and various controls to support the user in their exploration of the graph. The control buttons are located at position (a) in the top-right corner of the web page. They consist of a call-to-action button to start and stop the visualization, and three buttons for additional functionality. (b) shows the name of the graph and a brief description. The canvas elements used for the visualization, consisting of the detail view (c) and the main view (d), are positioned in the middle of the web page. By changing the value of the depth slider (e), the user is able to change the level-of-detail in the main view. A depth of one resembles the most abstracted visualization of the graph, while the maximal depth level shows the original graph data. The page slider (f) at the bottom of the web page controls the graph sequence, with the starting frame at the left and the last frame of the series on the right side.

An overview of the control buttons in use is shown in figure 5.2. They support the following controls:

(a) Clears the selection mark and all hierarchical marks from all nodes.

(b) Enables attribute visualization. In this state, the user can select individual nodes by using the right mouse button and view the attributes table.

Figure 5.2: The available control buttons in the DaGO application. (a) Clear selection from nodes, (b) enable attribute visualization on right mouse click, (c) enable hierarchy visualization on right mouse click, (d) enable auto-focus on right mouse click, (e) disable auto-focus on right mouse click, (f) start rendering and (g) stop rendering.

(c) Enables hierarchy visualization. In this state, the user can select a node to create and show hierarchical relations between aggregated elements.

(d) Enable auto-focus and link both views. By clicking the right mouse button in one of the views, the camera of the other view automatically moves to the clicked position. This way, fast navigation between frames is possible.

(e) Disable auto-focus.

(f) Starts the visualization of the selected graph and shows the first page of the sequence.

(g) Stops the current visualization and shows a select element for graph selection.

The buttons (b) and (c), (d) and (e), and (f) and (g) from pairs and resemble opposing states. They are shown in an alternating manner, so whenever one state is active, the button to activate the opposing state is shown and vice versa.

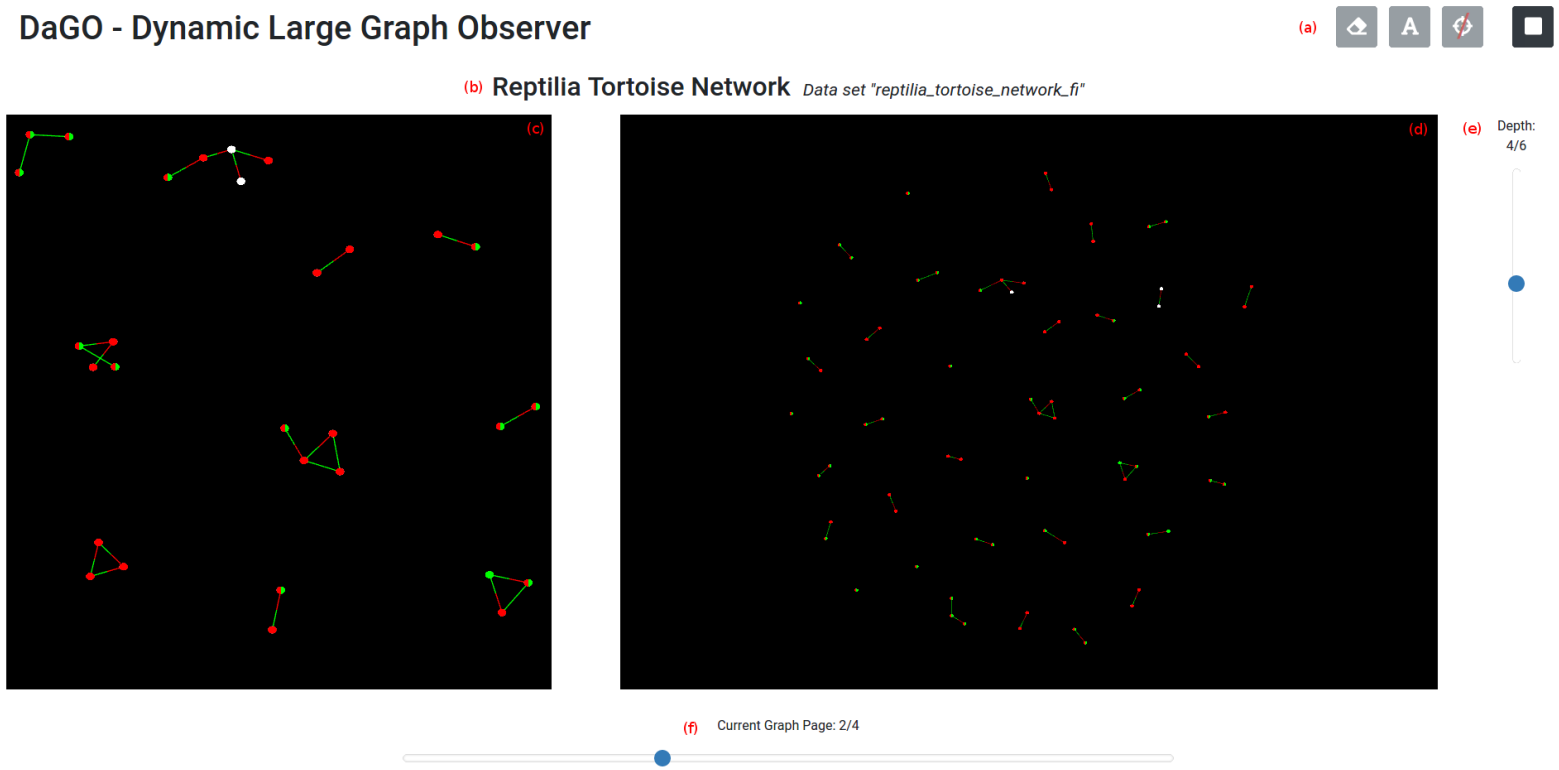


Figure 5.3: The interface of our DaGO application during the visualization of the reptilia tortoise network data set. (a)Control buttons, (b) name and description of the visualized graph, (c) detail view, (d) main view, (e) depth selector, (f) graph sequence selector.

CHAPTER 6

Results

To evaluate the data generation process and the running time of DaGO, we used the *RegularGraphs* benchmark and the *insecta-ant-colony-6* data set, which contains graphs of varying size and characteristics, extracted from both real-world applications and synthetic generators. The data sets consists of graphs with a size of up to 61,484 total elements. These graphs are taken from the TU Dortmund Benchmark Data Set [TU 20] and from the Network Data Repository [Ros15].

6.1 Data Analysis

A core concept of our approach is to pre-process graph data to generate and store a level-of-detail structure. A quadtree data structure is used to create a compound graph for each depth level. This technique, as described in section 4.4.2, enables us to fetch data of a specific detail level for visualization, which reduces the overall workload in the web frontend. This section evaluates the generation process of the level-of-detail structure by analyzing the created amount of data, as well as the number of generated depth levels and there rate of aggregation.

6.1.1 Generated data

Table 6.1 compares the data of the original graphs with the total amount of generated data required to store all level-of-details individually. Additionally, the number of depth levels created by the quadtree structure for each graph is reported as well. While the number of nodes in the original graph data ranges between 34 and 15,606, the number of edges varies between 78 and 45,878 . On the contrary, between 140 and 121,750 nodes and 344 to 381,311 edges are generated during the aggregation process. On average, the number of node elements increases by 655%, while the total number of edges rises by 670%. These data elements create quadtree structures of a depth ranging between 5 and 19.

Graph info	Nodes	Edges	G. nodes	G. edges	Depth
3elt	4,270	13,722	26,752	84,943	11
4elt	15,606	45,878	121,750	381,311	14
516	516	729	2,195	3,622	8
4970	4,970	7,400	27,633	46,599	11
add32	4,960	9,462	36,775	76,281	13
crack	10,240	30,380	42,311	136,374	10
cylinder_rnd_010_010	97	178	245	560	5
cylinder_rnd_032_032	985	1,866	3,723	8,419	8
cylinder_rnd_100_100	9,497	17,941	48,975	106,745	11
data	2,851	15,093	16,577	89,376	11
dg_617_part	341	797	1,206	3,243	7
esslingen	2,075	4,769	11,538	29,857	11
flower_001	210	3,057	729	9,619	7
flower_005	930	13,521	4,737	63,960	10
grid400_20	8,000	15,580	47,488	102,295	12
Grid_20_20	400	760	1,427	3,168	7
Grid_20_20_doublefolded	397	760	1,415	3,223	7
Grid_40_40	1,600	3,120	3,968	9,791	7
Grid_40_40_doublefolded	1,597	3,120	7,532	16,312	9
Grid_40_40_singlefolded	1,599	3,120	7,370	16,350	9
grid_rnd_032	985	1,834	2,113	4,865	6
grid_rnd_100	9,497	17,849	30,024	67,775	9
karateclub	34	78	140	344	6
protein_part	417	511	2,573	3,527	10
rna	363	468	1,173	1,814	7
sierpinski_04	123	243	408	914	6
sierpinski_06	1,095	2,187	3,655	8,184	8
sierpinski_08	9,843	19,683	84,843	177,462	15
snowflake_A	98	97	320	393	6
snowflake_B	971	970	8,435	9,077	14
snowflake_C	9,701	9,700	84,226	92,137	16
spider_A	100	160	320	504	6
spider_B	1,000	1,600	6,827	11,090	12
spider_C	10,000	22,000	80,405	170,644	15
tree_06_03	259	258	992	1,172	7
tree_06_04	1,555	1,554	11,241	12,631	12
tree_06_05	9,331	9,330	98,308	116,823	19
ug_380	1,104	3,231	6,733	22,637	11
uk	4,824	6,837	30,510	48,326	12

Table 6.1: RegularGraphs: Comparison of the original data and the generated data, including the depth level of the quadtree structure.

Depth Level	Nodes	Edges	Abstraction
1	4	14	99.9707%
2	15	73	99.8569%
3	53	293	99.4373%
4	201	1173	97.7653%
5	761	4482	91.4726%
6	2884	15919	69.4181%
7	9768	38786	21.0299%
8	14618	45384	2.4104%
9	15458	45814	0.3448%
10	15576	45870	0.0618%
11	15598	45873	0.0211%
12	15603	45875	0.0098%
13	15605	45877	0.0033%
14	15606	45878	0%

Table 6.2: Breakdown of the individual depth levels of the quadtree structure, created during the processing of the *4elt* graph.

6.1.2 Quadtree analysis

The significant rise of total data volume by over 650% on average has been analyzed in more detail. Table 6.2 shows an in-depth analysis of the quadtree structure created during the pre-processing of the *4elt* graph. It allows comparisons between individual depth levels and their respective amounts of abstraction. Each depth level consists of a specific amount of meta-elements, which aggregate the original graph. The degree of aggregation is illustrated in the "Abstraction" column. The fewer elements are used to visualize the graph, the higher the aggregation value becomes. Since the last depth level contains the original graph data, an abstraction value of 0% is set.

From the data, it appears that the last six depth levels do not differ much in regards to their aggregation values. Level 9 to 13 contain overall 307, 149 elements, which corresponds to 61, 06% of the total amount of data created by the quadtree structure. Therefore, more than half of the overall data is used to create multiple level-of-details, aggregating less than half a percent of the total amount of elements.

The same observation applies to the upper depth levels 1 to 3. Though, due to the already high degree of aggregation, only 452 element (0.09%) would be removed. Thus leading to the conclusion that the upper levels can be included in the quadtree structure if needed, without drastically affecting the overall amount of data.

The above observations lead to the conclusion that the quadtree structure should not be stored in the database on a 'one-to-one' basis. A better approach to reduce the total amount of data would be to ignore certain depth levels, which have only a small impact on the abstraction level.

Graph info	Pre	Lay.	Post	Qt	Stor.	Total
3elt	915.07	1,180.15	56	48	212.40	214.60
4elt	2,819.33	3,916.76	181	169	1010.90	1,017.99
516	79.46	248.90	7	3	11.15	11.49
4970	807.06	948.62	68	35	151.77	153.62
add32	1,028.67	2,018.48	63	41	228.05	231.20
crack	2,615.29	3,644.07	133	117	365.94	372.45
cyl_rnd_010_010	20.69	171.16	2	1	1.74	1.94
cyl_rnd_032_032	183.95	371.26	14	7	23.79	24.37
cyl_rnd_100_100	1,510.77	2,714.05	120	89	318.21	322.65
data	571.47	1,325.42	30	46	248.69	250.67
dg_617_part	50.14	322.18	5	4	9.31	9.69
esslingen	370.88	2,226.37	26	20	85.31	87.96
flower_001	204.29	572.77	5	23	27.81	28.62
flower_005	416.87	841.72	13	39	180.48	181.79
grid400_20	1,740.72	1,956.64	120	98	305.17	309.09
Grid_20_20	76.87	295.24	9	4	9.56	9.94
Grid_20_20_df.	64.54	414.24	6	3	9.58	10.07
Grid_40_40	332.65	588.46	31	17	29.09	30.06
Grid_40_40_df.	241.42	454.23	18	10	45.81	46.54
Grid_40_40_sf.	314.60	662.26	21	13	48.71	49.72
grid_rnd_032	170.50	291.43	14	6	15.31	15.79
grid_rnd_100	1,857.46	2,797.00	125	74	205.07	209.93
karateclub	10.35	928.52	4	0	0.97	1.91
protein_part	64.43	384.56	7	2	12.16	12.62
rna	51.60	319.56	5	2	6.43	6.81
sierpinski_04	36.69	324.74	6	4	4.21	4.58
sierpinski_06	168.17	281.62	19	9	24.35	24.83
sierpinski_08	1,810.32	1,639.13	146	105	570.07	573.77
snowflake_A	19.30	402.33	14	2	1.73	2.17
snowflake_B	161.85	617.45	14	5	36.09	36.89
snowflake_C	1,510.38	30,824.56	191	93	370.40	403.02
spider_A	26.93	315.96	4	1	2.56	2.91
spider_B	149.34	267.33	14	8	39.74	40.18
spider_C	1,427.60	5,195.00	104	86	601.17	607.99
tree_06_03	35.44	628.21	7	2	5.23	5.90
tree_06_04	243.42	750.63	28	12	53.65	54.69
tree_06_05	2,112.46	6,275.61	193	120	463.60	472.30
ug_380	210.98	6,704.74	16	11	60.81	67.75
uk	595.52	794.08	61	27	212.77	214.25

Table 6.3: RegularGraphs: Running time of individual parts of the pre-processor and layout service during the creation of a new graph sequence element. The measurements for the steps pre-processing, layout, post-processing, and quadtree are in milliseconds, while the measured time for the storage and total time columns are in seconds.

6.2 Performance Analysis

This section covers the performance analysis of the two core functionalities of our system: Creation of a new graph and visualization of an existing graph. Performance has been measured by tracking the execution time of interesting parts of the code and individual components of the system.

The experiments on DaGO were executed on a Notebook equipped with a Intel Core i7-6700HQ CPU @ 2.60GHz, 8GM RAM and a Nvidia GeForce GTX 960M graphics card, running Ubuntu 16.04.6 LTS and Mozilla Firefox 71.0 (64-bit).

6.2.1 Creation of a new graph

The process of creating a new graph is separated into two parts: processing static graphs and dynamic graphs. This separation allows analyzing individual steps included in the respective task in more detail.

Static graphs

In Table 6.3, we report the timing breakdown for the five major phases of the creation process, as well as the total time of the generation process. The following steps are described in each column:

- **Pre-process (Pre-p.)** in **milliseconds**:
  Wraps all preparation steps of the pre-processor before a layout of the graph can be created. These include fetching the original graph file from the cache, parsing the input file, transforming the data into a format required by layout service, and storing the input file for the layout process in the cache.
- **Layout (Lay.)** in **milliseconds**:
  This column lists the total time required by the layout service to create a layout for the given graph.
- **Post-process (Post-p.)** in **milliseconds**:
  Wraps all steps to convert the results of the layout service into a structure that can be used to create a quadtree structure. It contains the steps of fetching the layout results from the cache, the parsing of the output file, and the conversion into a generic data structure.
- **Quadtree (Qt)** in **milliseconds**:
  Lists the measured time to create and fill the quadtree data structure.
- **Storage (Stor.)** in **seconds**:
  Summarizes the time to create a new graph sequence element in the database, to extract and store all data elements from the quadtree structure for the respective depth level.

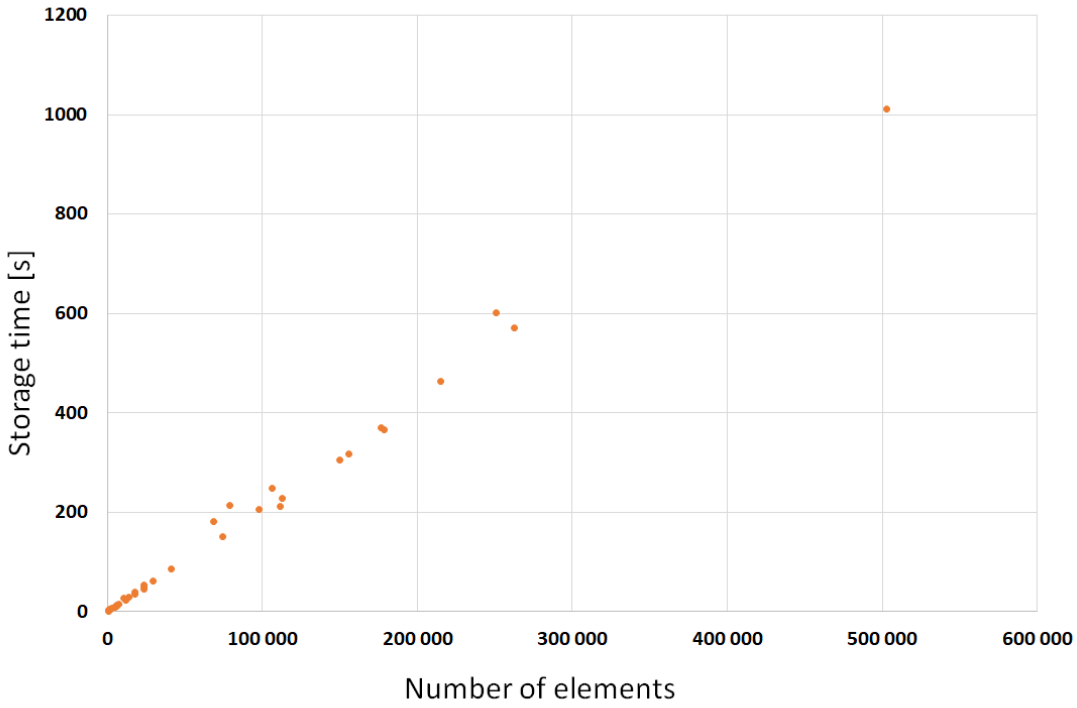


Figure 6.1: Quadtree storage time per total generated elements during the creation of a new static graph.

- **Total** in **seconds**:
  The total time to layout, pre-process and store the respective graph sequence element.

The overall processing time of the graphs shown in table 6.3 ranges between 1.8 seconds and 16.97 minutes. From the data, it appears that the quadtree storage time constitutes a particularly expensive operation. By taking up, on average, 94.97% of the overall processing time, this step represents the major bottleneck of the creation process. On the other hand, the layout step holds, on average, 4.34%, while the pre-layout, post-layout, and quadtree creation steps take up together only 0.68% of the total execution time.

The graph 6.1 shows a linear correlation between the storage time of the quadtree and the total number of generated elements during the creation of a new graph. We can therefore assume that the performance would improve respectively when the total number of elements to persist in the database is reduced.

Dynamic graphs

Insecta-ant-colony-6 is a real-world animal interaction network data sets. It shows interaction data of an ant colony from published studies of wild, captive, and domesticated animals, and consists of recordings of 42 consecutive days.

Table 6.4 shows a custom graph sequence made out of 5 days from the original *insecta-ant-colony-6* dataset. The following steps are described in each column:

- **Change**:
  This column lists the total amount of changed elements in the last two sequence elements of the graph. The number contains both created and deleted elements.
- **Fetch**:
  This step lists the measured time to fetch all node and edge elements of the highest depth level from the last sequence element and its predecessor.
- **Extraction (Extr.)**:
  Wraps all steps required to extract all elements which have been newly created or deleted.
- **Update**:
  Summarizes the time to update the lifetime in the database of all extracted node and edge elements and their respective meta-elements of the last two sequence elements of the graph.
- **Total**:
  The total time for the sequence service to add lifetime information to the respective graph sequence elements.

One can see that the fetch and update steps represent the performance-intensive operations of the sequence service. On average, the fetch step takes up 42%, while the update procedure holds 51% of the total processing time. On the other hand, the extraction step takes up only 7% on average of the service's execution time.

It can be seen that the performance of both expensive steps correlates with the total number of processed elements. Since the fetch procedure is based on original data, reducing the total amount of data to represent the level-of-detail structure will not affect its runtime. However, the update step depends on the total number of elements and created depth levels. By reducing the amount of data generated via the pre-processing service, the performance of this step could be improved, respectively.

Day	Node	Edge	Change	Fetch [s]	Extr. [s]	Update [s]	Total [s]
1	164	10731	0	0	0	0	0.001
2	164	10272	16317	19.485	4.722	18.137	42.346
12	143	6625	14002	15.555	3.097	16.322	34.977
22	118	4885	5571	12.555	1.439	15.857	29.853
32	91	2817	7165	9.191	0.938	15.186	25.318

Table 6.4: insecta-ant-colony_6: Running time of individual parts of the pre-processor and layout service during the creation of a new graph sequence element.

6.2.2 Visualization of an existing graph

Table 6.5 shows the timing breakdown of both services constituting the visualization process of a graph sequence. The following steps are described in each column.

- **Backend total**:
  Wraps all steps of the backend service required to provide the requested graph data to the web frontend. Since the backend service acts as a data access API in the visualization process, the total processing time represents mostly database fetch operations.
- **Set data**:
  This column lists the total time required by the web frontend to prepare the fetched data provided by the backend service for rendering. It contains the steps to transform graph elements into two-dimensional objects and to prepare them for rendering.
- **Render time**:
  Lists a snapshot of the first measured time to render an individual frame of the respective graph.

The total processing time of the backend service shown in table 6.5 ranges between 407 and 1,908 milliseconds and represents by far the most time-consuming operation in the visualization process. On the other hand, the set data step holds 2 to 557 milliseconds, and the render time takes up between 1 to 9 milliseconds.

Graph info	Backend total [ms]	Set data [ms]	Render time [ms]
3elt	887	209	5
4elt	1908	557	9
516	507	12	2
4970	734	97	2
add32	606	119	2
crack	1030	378	4
cyl_rnd_010_010	453	3	2
cyl_rnd_032_032	574	29	2
cyl_rnd_100_100	1026	224	4
data	666	184	2
dg_617_part	426	10	2
esslingen	630	61	3
flower_001	558	53	1
flower_005	741	211	4
grid400_20	1208	198	4
Grid_20_20	440	13	2
Grid_20_20_df.	510	13	2
Grid_40_40	492	40	2
Grid_40_40_df.	467	44	1
Grid_40_40_sf.	555	45	2
grid_rnd_032	456	25	2
grid_rnd_100	1091	226	3
karateclub	408	2	1
protein_part	434	7	1
rna	407	8	4
sierpinski_04	489	5	3
sierpinski_06	479	34	4
sierpinski_08	920	281	4
snowflake_A	462	2	2
snowflake_B	537	19	2
snowflake_C	632	139	4
spider_A	431	7	5
spider_B	496	30	3
spider_C	941	286	6
tree_06_03	574	6	3
tree_06_04	477	27	2
tree_06_05	756	130	2
ug_380	520	52	6
uk	585	97	5

Table 6.5: RegularGraphs: Running time of individual parts of the backend service and web frontend during the visualization of a graph.

Figure 6.2 compares both steps of the web frontend with the total number of processed elements. (a) shows a linear correlation between the set data step and the number of elements. This progression is expected since every single data element has to be processed individually. On the other hand, by sending only single values to represent graph elements, the total size of data sent between the services is reduced. Additionally, this allows the frontend to manipulate the creation process and, therefore, to customize the visualization. With an average of 100 milliseconds and a maximum value of roughly half a second, this step takes up a reasonable waiting time when compared to the advantages resulting from moving the object creation process to the frontend. (b) shows a rough overview of the render time of each graph. It should be noticed that these values fluctuated significantly during the evaluation, and no distinct correlation could be identified. However, no graph exhibits a render time of more than 11 milliseconds. Therefore, an interactive visualization with 60 frames per second could be achieved consistently.

Hierarchy and selection

Most of the workload regarding the process of visualizing a hierarchy structure is executed by the backend service. First, successor and predecessor nodes of the node element specified by the web frontend are identified, and a list containing their quadtree ids is created. The performance of this process depends on the total number of elements representing a graph and its level-of-detail structure, as well as the total number of depth levels. However, the list can be constructed rather efficiently. During the evaluation of DaGO, a list consisting of $21,851$ elements was generated in 25 milliseconds.

Last, all existing elements of the specified graph sequence with a quadtree id contained in the generated list, are fetched from the database. The performance of this process depends on the number of requested elements, as well as on the total number of generated elements for the respective graph sequence element. During the evaluation of DaGO, the execution time of the database request ranged between milliseconds and half a minute.

On the other hand, the time to select an individual element in the web frontend could not be measured accurately due to the resolution of the used timer. However, selecting a quarter of the elements at the deepest depth level of the *4elt* graph took 6 milliseconds in total.

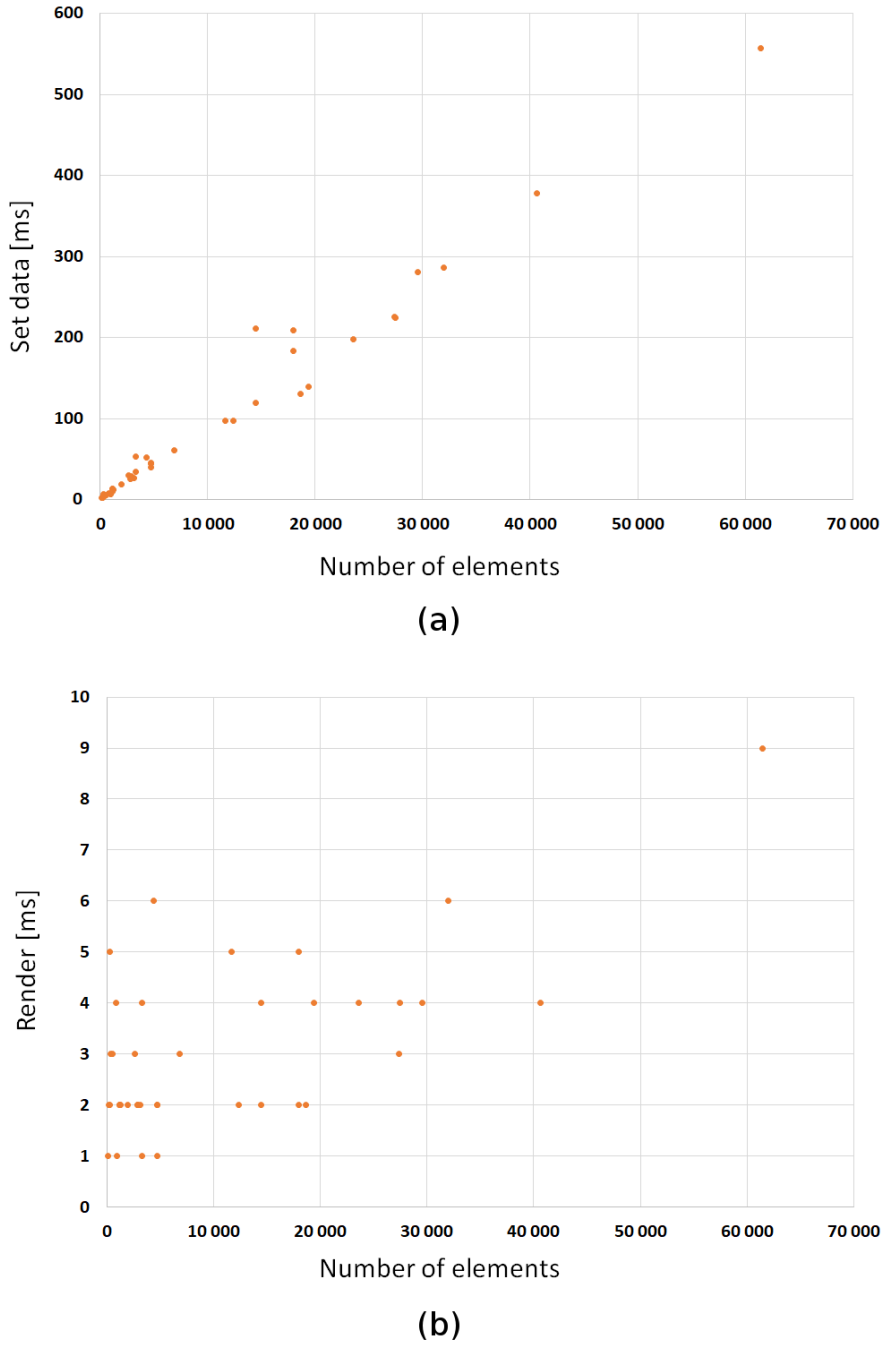


Figure 6.2: (a) Time to set data per total number of elements. (b) Render time per total number of elements.

CHAPTER 7

Conclusion and Future Work

In this work, we have presented a new approach to visualize large dynamic node-link diagrams interactively in a standard web browser. The generation of meta node and meta edge elements allows the visualization of large graphs at different levels of detail, each layer represented as a separate compound graph. Hierarchy is visualized on-demand through color to show information about the generated aggregation within the level-of-detail structure. This allows analyzing connections across individual aggregation layers. For fully dynamic graphs, the visualization of the graph's structural development throughout the whole sequence is of particular importance. Thus, newly created and soon-to-be deleted elements are highlighted by color to enable the communication of overall changes to the user in an animated approach. Additionally, the interactive techniques brushing and linking, as well as panning and zooming, are used to display, navigate, explore, and analyze graph data. By selecting individual node elements, domain-specific attributes associated with the selected component are listed.

The methods are designed to move performance and time-intensive operations to the backend. By precomputing a layout and a level-of-detail structure for each graph, the results can be reused by all users of the system, while the role of the web frontend is reduced to the rendering of the visualization. By using methods optimized for fast processing in WebGL, we allow interactive rates at above 60 frames-per-second when using standard graphics hardware.

From our main research question **What are practical implications of interactive visualizations of large dynamic graphs with modern web-based technology?**, we derived the questions **Q1** how a large dynamic graph, consisting of thousands of elements, can be visualized in a standard web browser and **Q2** how real-time interactive exploration and visualization techniques can be implemented with web-based technologies.

A prototype application named DaGo was built based on the learnings from the literature research and on the stated hypotheses and evaluated to validate these presumptions.

It could be shown that the WebGL API, in combination with WebAssembly, is capable of continuously rendering interactive visualizations of graphs consisting of 60,000 components at more than 60 frames-per-second. The non-linear correlation between the render process and the total number of processed elements seem to indicate that even larger graphs could be rendered in real-time.

Furthermore, it could be shown that by pre-processing graphs and extracting and storing information prior to the actual visualization step, the performance of the visualization system is not limited by the client's hardware but by the performance of the backend and mostly the used storage system. Thus, the client's memory and processing power can be applied to various interactive visualization methods, which enables real-time exploration of a graph. Furthermore, the approach allows the client to use older hardware without compromising the interactivity of the visualization.

It can, therefore, be concluded that this new method represents a balanced approach for use-cases regarding a client with a consistent connection to the worldwide web. On one hand, large dynamic graphs of a size smaller than a hundred thousand elements can be visualized interactively in real-time on devices equipped with a modern web browser. On the other hand, the distributed micro-service architecture does not require expensive compute clusters or extensive server power.

Although DaGO is able to visualize large dynamic graphs interactively in a browser it also encloses several limitations which can be targeted in future work:

- Graph data is stored in the database, and a high number of read, write and update operations are executed during the pre-processing and visualization steps. The evaluation has shown that these operations represent a significant bottleneck in the performance of many services. Thus different types of databases could be analyzed regarding there performance. Alternatively, bulk operations could be used to improve the overall speed of the write and update procedures.
- Since all clients of the system request the same graph data, cache reoccurring information can reduce the number of database operations. Furthermore, the following graph elements of a requested sequence could be loaded from the database and stored in the caching system in advance to prepare for future requests. Thus, a fast caching system could improve the performance of the visualization process.
- The evaluation process has shown that the quadtree structure can be used to create a level-of-detail structure. However, it was shown that some depth levels do not contribute a sufficient degree of abstraction, but instead increase the overall number of generated elements. Therefore, by post-processing the quadtree structure and only using depth levels with a reasonable degree of aggregation, the total number

of generated elements and the required storage space could be reduced drastically. Additionally, the linear correlation between the overall performance and the total number of processed elements in most services would suggest that the reduction of generated elements will likely lead to performance improvements as well.

- The system has only been tested on a single computer, using the loopback network interface for transportation. Thus, the speed and bandwidth of actual networks could become a bottleneck for larger graphs in real-world applications. Further analysis of data size and possible optimizations could be done in future work to reduce loading time.
- Since a modern web browser is the only requirement for a client to use the application, further tests and optimized web interfaces for mobile devices could be realized.

APPENDIX A

Download and Installation

1. Download and install the latest version of Docker `https://docs.docker.com/install/`.

2. Download and install the latest version of Docker-Compose `https://docs.docker.com/compose/install/`.

3. Clone the Git repositories listed below. Alternatively, you can download the ZIP-files with a web browser of your choice and extract the packages afterward. It should be mentioned that all repositories should be placed in the same folder next to each other.

4. A Unix system is required to execute the created control script.

The DaGO application has been split up into seven individual repositories. The following list contains the URL and a short description for each repository:

- **Frontend**:
  The web UI of this project with all its modules.
  `https://bitbucket.org/matthias_reisacher/master-thesis-frontend/src/master/`

- **Backend**:
  This is the main backend service that receives all calls from the frontend.
  `https://bitbucket.org/matthias_reisacher/master-thesis-backend/src/master/`

- **Layout**:
  This is the layout service published by Crnovrsanin et al. [CT15] including a wrapper for gRPC and Redis.
  `https://bitbucket.org/matthias_reisacher/master-thesis-backend-layout/src/master/`

- **Preporcessor**:
  This micro-service converts all files into a generic format, sends the data to the layout service and processes the positioned graph elements afterward.
  `https://bitbucket.org/matthias_reisacher/master-thesis-backend-preprocessor/src/master/`

- **Sequencer**:
  This micro-service takes the individual pages created by the preprocessor and compares each one with its predecessor to add information about the lifetime to each element.
  `https://bitbucket.org/matthias_reisacher/master-thesis-backend-sequencer/src/master/`

- **Environment**:
  This repository contains all scripts to start the system, the database structure, front proxy mappings, and the container-based infrastructure.
  `https://bitbucket.org/matthias_reisacher/master-thesis-environment/src/master/`

- **Protobuf**:
  This repository contains all gRPC message and service descriptions and is required by all other repositories.
  `https://bitbucket.org/matthias_reisacher/master-thesis-protorepo/src/master/`

APPENDIX B

Configure and Run

Multiple shell scripts are available to control the DaGO application. The following steps have to be executed to start the system:

1. Open a terminal and navigate into the *master-thesis-environment* folder.

   ```
   cd path/to/repositories/master-thesis-environment/
   ```

2. Start the *setup* routine, which verifies the dependencies, builds the docker images and initializes the database. This step can take a couple of hours when executed the first time.

   ```
   ./script/setup.sh
   ```

3. Start the *run* routine, which generates and starts the docker containers and opens the combined log-file.

   ```
   ./script/run.sh
   ```

4. Open a web browser and visit the URL `http://localhost:8080` to load the DaGO frontend.

Optional: Cleanup the system by removing all created images and containers.
Note: Docker volumes must be deleted manually!

```
./script/cleanup.sh
```

List of Figures

List of Tables

List of Algorithms

Bibliography

[AA17] Liotta G. Montecchiani F. Arleo A., Didimo W. Large graph visualizations using a distributed computing platform. *Information Sciences*, 381:124–141, 2017.

[BBD09] Fabian Beck, Michael Burch, and Stephan Diehl. Towards an aesthetic dimensions framework for dynamic graph visualisations. In *13th International Conference Information Visualisation*, pages 592–597. IEEE, 2009.

[BF14] Diehl S. Weiskopf D. Beck F., Burch M. The state of the art in visualizing dynamic graphs. *EuroVis STAR*, 2:1–21, 2014.

[BJ76] Murty U.S.R. Bondy J.A. *Graph theory with applications*. Citeseer, 5. edition, 1976.

[BM17] Weiskopf D. Burch M., Hlawatsch M. Visualizing a sequence of a thousand graphs (or even more). *Computer Graphics Forum*, 36(3):261–271, 2017.

[CQ07] Weiwei Cui and Huamin Qu. A survey on graph visualization. *PhD Qualifying Exam (PQE) Report, Computer Science Department, Hong Kong University of Science and Technology, Kowloon, Hong Kong*, 2007.

[CT15] Ma K.-L. Crnovrsanin T., Chu J. An incremental layout method for visualizing online dynamic graphs. In *International Symposium on Graph Drawing and Network Visualization*, pages 16–29. Springer, 2015.

[CW08] Qu H.-Wong P. C. Li X. Cui W, Zhou H. Geometry-based edge clustering for graph visualization. *IEEE Transtions on Visualization and Computer Graphics*, 14(6):1277–1284, 2008.

[Doc19] Docker Inc. Docker Homepage. `https://www.docker.com/`, 2019. Online; accessed 10 December 2019.

[EF09] Niklas Elmqvist and Jean-Daniel Fekete. Hierarchical aggregation for information visualization: Overview, techniques, and design guidelines. *IEEE Transactions on Visualization and Computer Graphics*, 16(3):439–454, 2009.

[Env19] Envoy Project Authors. Enovy Homepage. https://www.envoyproxy.io/, 2019. Online; accessed 28 December 2019.

[GFC04] Mohammad Ghoniem, J-D Fekete, and Philippe Castagliola. A comparison of the readability of graphs using node-link and matrix-based representations. In *IEEE Symposium on Information Visualization*, pages 17–24. IEEE, 2004.

[Goo19] Google Inc. gRPC Documentation. https://grpc.io/docs/guides/, 2019. Online; accessed 21 December 2019.

[HA15] Antoine Hinge and David Auber. Distributed graph layout with Spark. In *Information Visualisation (iV), 2015 19th International Conference on*, pages 271–276. IEEE, 2015.

[HC14] Fabrikant S.I.-Klein T.R. Telea A.C. Hurter C., Ersoy O. Bundled visualization of dynamicgraph and trail data. *IEEE Transactions on Visualization and Computer Graphics*, 20(8):1141–1157, 2014.

[Hem06] Anil Hemrajani. *Agile Java development with spring, hibernate and eclipse*. Sams publishing, 2006.

[HRS+17a] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. Bringing the web up to speed with WebAssembly. *ACM*, 52(6):185–200, 2017.

[HRS+17b] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *ACM SIGPLAN Notices*, volume 52, pages 185–200. ACM, 2017.

[JI13] Zimmer B. Jusufi I., Kerren A. Multivariate network exploration with JauntyNets. In *Information Visualisation (IV), 2013 17th International Conference*, pages 19–27. IEEE, 2013.

[KAF+08] Daniel Keim, Gennady Andrienko, Jean-Daniel Fekete, Carsten Görg, Jörn Kohlhammer, and Guy Melançon. Visual analytics: Definition, process, and challenges. In *Information visualization*, pages 154–175. Springer, 2008.

[Khr19a] Khronos Group. WebGL Extension Registry. https://www.khronos.org/registry/webgl/extensions/, 2019. Online; accessed 9 December 2019.

[Khr19b] Khronos Group. WebGL Overview. https://www.khronos.org/webgl/, 2019. Online; accessed 9 December 2019.

[Kol09] Eric D. Kolaczyk. *Statistical Analysis of Network Models*. New York: Springer, 2009.

[KSKB17] Sándor Királya, Szilveszter Székelyb, Roland Királyc, and Tamás Ballad. Some aspects of using RPC. In *Proceedings of the 10th International Conference on Informatics and Systems*, pages 145–156. ACM, 2017.

[LL13] Scheidegger C. Lins L., Klosowski J.T. Nanocubes for Real-Time Exploration of Spatiotemporal Datasets. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2456–2465, 2013.

[LPP+06] Bongshin Lee, Catherine Plaisant, Cynthia Sims Parr, Jean-Daniel Fekete, and Nathalie Henry. Task Taxonomy for Graph Visualization. In *Proceedings of the 2006 AVI Workshop on BEyond Time and Errors: Novel Evaluation Methods for Information Visualization*, pages 1–5. ACM, 2006.

[LZ13] Heer J. Liu Z., Jiang B. imMens: Real-time Visual Querying of Big Data. In *Computer Graphics Forum*, pages 421–430. Wiley Online Library, 2013.

[MKI14] Nicholas D Matsakis and Felix S Klock II. The Rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[PA15] Burch M.-Pfannkuch T. Ertl T. Panagiotidis A., Reina G. Consistently GPU-Accelerated Graph Visualization. In *Proceedings of the 8th International Symposium on Visual Information Communication and Interaction*, pages 35–41. ACM, 2015.

[PA18] Auber D. Perrot A. Cornac: Tackling Huge Graph Visualization with Big Data Infrastructure. *IEEE transactions on big data*, 2018.

[Par12] Tony Parisi. *WebGL: up and running.* O'Reilly Media, Inc., 2012.

[PV14] Marco Panunzio and Tullio Vardanega. A component-based process with separation of concerns for the development of embedded real-time software systems. *Journal of Systems and Software*, 96:105–121, 2014.

[PV15] Telea A. Peysakhovich V., Hurter C. Attribute-driven edge bundling for general graphs with applications in trail analysis. In *Computer graphics forum*, pages 39–46. IEEE, Pacific Visualization Symposium, 2015.

[RH13] Bach B.-Henry-Riche N. Pietriga E. Romat H., Appert C. Animated Edge Textures in Node-Link Diagrams: a Design Space and Initial Evaluation. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 187–200. ACM, 2013.

[Ros15] Rossi R., Ahmed N. The Network Data Repository with Interactive Graph Analytics and Visualization. `http://networkrepository.com`, 2015. Online; accessed 03 January 2020.

[Sam84] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, 16(2):187–260, 1984.

[SH15] Weiskopf D. Schmauder H., Burch M. Visualizing Dynamic Weighted Digraphs with Partial Links. In *IVAPP*, pages 123–130, 2015.

[Str00] Bjarne Stroustrup. *Die C++-Programmiersprache*. Pearson Deutschland GmbH, 2000.

[Tar19] Tarik Crnovrsanin. Online Dynamic Graph Layout. `https://bitbucket.org/turokhunter/online-dynamic-graph-layout/src/master/`, 2019. Online; accessed 30 December 2019.

[TU 20] TU Dortmund. Benchmark data set information. `http://ls11-www.cs.tu-dortmund.de/staff/klein/gdmult10`, 2020. Online; accessed 03 January 2020.

[Tur14] James Turnbull. *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.

[TY08] Patel J.M. Tian Y., Hankins R.A. Efficient aggregation for graph summarization. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 567–580. ACM, 2008.

[VdZM16] Alexandru T. Van der Zwan M., Valeriu C. CUBu: Universal real-time bundling for large graphs. *IEEE transactions on visualization and computer graphics*, 22(12):2550–2563, 2016.

[VLKS+11] Tatiana Von Landesberger, Arjan Kuijper, Tobias Schreck, Jörn Kohlhammer, Jarke J van Wijk, J-D Fekete, and Dieter W Fellner. Visual analysis of large graphs: state-of-the-art and future research challenges. *Computer graphics forum*, 30(6):1719–1749, 2011.

[W3C19] W3C Community. WebAssembly Documentation. `https://webassembly.org/`, 2019. Online; accessed 29 December 2019.

[War12] Colin Ware. *Information visualization: perception for design*. Elsevier, 2012.

[WC08] Huamin Q. Pak C.W. Xiaoming L. Weiwei C., Hong Z. Geometry-based edge clustering for graph visualization. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1277–1284, 2008.

[ZM12] Deussen O. Strobelt H. Zinsmaier M., Brandes U. Interactive level-of-detail rendering of large graphs. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2486–2495, 2012.