Developing an Interactive Visualization Tool for Pangenome Graphs

Yulian Martynovsky

A thesis presented for the degree of
Master of Science

Heinrich Heine University Düsseldorf
Germany
03th October, 2022

Acknowledgments

I express my gratitude to my assessors Prof. Dr. Gunnar Klau and Prof. Dr. Tobias Marschall for providing this interesting topic. I owe special thanks to Philipp Spohr for being a very helpful and available supervisor, and answering my many questions and taking a lot of time for weekly meeting and feedback on my work. Further, I thank Max Ried for setting up with the website and providing support. I thanks Jonas Weber and Matthias Häußler for supporting my writing and thinking processes and providing nice work environments.

Abstract

In understanding and interpreting genomic data, visualization plays an important role. To analyze sequencing read data, the reads are often aligned to a reference genome. By doing so the position of each read in the genome is identified. Since the alignment process yields complex data, there is a need to manually review the data and therefore there is a need for tools which enable this. If multiple references genomes are available, they can be combined as a non-linear reference. Such a reference can be encoded in the form of a pangenome graph. We developed GIraph, a tool to visualize such graphs. It allow the user to interact with the visualized non-linear reference and gain insight into the structure of the graph. Additionally, read data can be incorporated into the visualization to to give a clearer overall view of how the reads align to the graph. In this thesis we present GIraph, a web application based on JavaScript and the React framework, for visualization of pangenome graphs and corresponding alignments to these graphs. We give an overview of the features of GIraph and provide some benchmarks for to showcase what graph sizes can be processed by GIraph.

Contents

1 Introduction

Reference genomes are used in the analysis of sequencing data. The sequenced reads are aligned to them to identify their likely presumed position in the genome. Linear reference genomes are commonly used to achieve this. But since they only represent one possible version of the genome, possible variations in the population are disregarded. This is a limitation for analyses that align sequenced reads to reference genomes. It leads to a biased interpretation of sequencing data due to reads not aligning or aligning at wrong positions if the original sequence comes from a different variant. This can then lead to issues in downstream analyses as such biases introduce errors to the mapping process [1, 2, 3]. In some cases, data on these variants is available [4] and can be incorporated into the analyses. This can be done by representing the reference as a graph instead of a linear string. Such graphs are called pangenome graphs or variation graphs. Tools like vg [5] or GraphAligner[6] can be used to align sequenced reads to pangenome graphs.

This creates a great amount of complex biological data, which leads to the need for tools that visualize this data in an intuitive way. For reads aligned to a linear reference, a widely used tool, IGV, already exists [7]. A tool widely used to visualize pangenome graphs is Bandage[8], which lacks the ability to display information about reads aligned to the graph. For reads aligned to a variation graph, there is also already a tool, namely SequenceTubeMaps [9]. Our goal is to develop a tool, which in addition to visualization of variation graphs and aligned reads, allows the user to interact with the graph.

1.1 Related Work

In this section, we will look at some existing tools that have similar functionalities. To identify features that our tool, GIraph, should have, we considered previously existing tools. IGV visualizes alignments to linear reference genomes, Bandage displays pangenome graphs and SequenceTubeMaps displays pangenome graph alignments. In Figures 1 and 2 we can see how alignments are visualized in IGV. As the reference genome is linear, i.e. a string, the reads can be listed at the position that they are mapped to. Reads that overlap are listed below each other. Reads are represented as gray tracks. Insertions are represented in blue and have a tooltip, that when clicked on to display the bases that are inserted. Deletions are white and display how many bases are deleted. If the track is gray the read matches the reference, otherwise it is either a deletion or insertion. Mismatches from the reference are displayed by writing the mismatching base on the read track. In Figure 3 we see the tooltip which appears by clicking of a read track. It displays additional information about the read. Furthermore, IGV allows for different filters to be applied to the reads. Moreover, there is a coverage track above the reads displaying the coverage at a specific position.

In Figure 4, we can see how Bandage [8] displays pangenome graphs. The layout is computed using the fast multipole multilevel method [10] and shows the structure of the pangenome graph. Bandage cannot be used to display information about reads aligning to these graphs.

In Figure 5, we see how SequenceTubeMaps visualizes the alignment. SequenceTubeMaps visualizes graph alignments, so the reference is not just written as a string. The nodes of the pangenome graph are represented as gray boxes and the sequence that the node encodes is displayed at the top of the box. Reads are, similar to IGV, tracks that follow a path through the nodes to which they align. While the track is in a node it is displayed aligned to the reference in the same way is in IGV. This allows the user to see how the reads are distributed in sections where, for example, there are SNPs and the read flow splits into multiple paths and then combines again into a single node. Here, deletions are displayed as grayed-out parts of the track, and insertions are marked with a "*" and have a tooltip that shows the inserted sequence. This means that it is not possible to visually see how long an insertion is without hovering over each insertion. SNPs are again displayed by writing the changed base on the read track. SequenceTubeMaps also features some filtering options like a mapping quality cutoff. Additionally, it can also display further information by color-coding mapping quality and forward and reverse strand reads. SequenceTubeMaps cannot display the variation graph without corresponding reads and expects a linear structure in the pangenome graph. The algorithm, as the name suggests, is based on the problem of drawing maps for subway networks, however, the authors do not describe how the algorithm works exactly [9]. The general problem of drawing such maps and rules that these maps want to fulfill is further described by Wolff et al. [11]. The goal is to find a layout such that the overarching structure, which in the case of subway maps is given by real-world coordinates, is retained. For the SequenceTubeMaps this overarching structure seems to be calculated using the aligned reads.

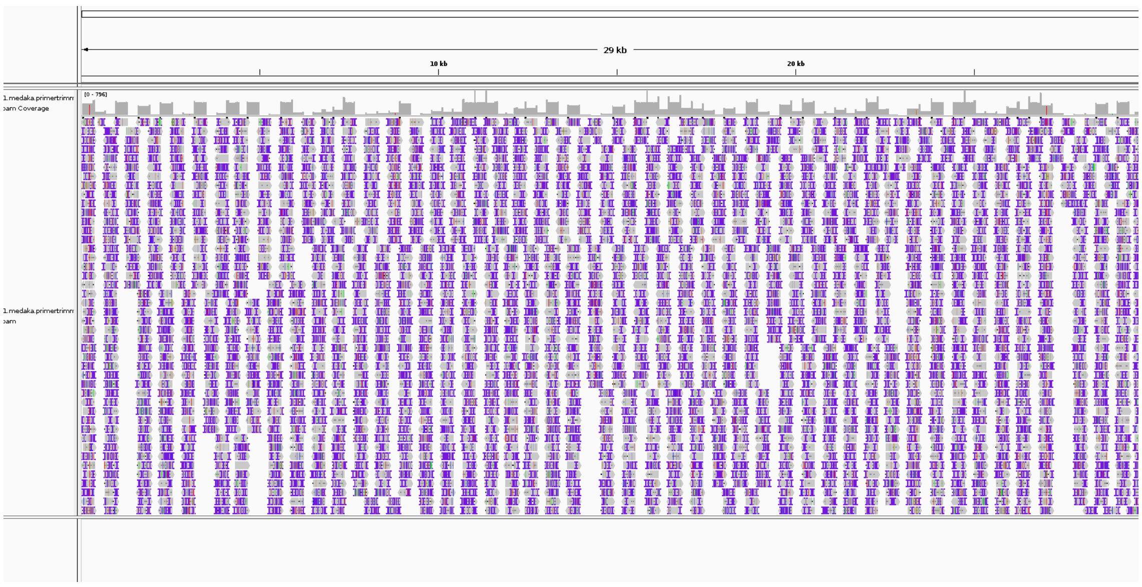


Figure 1: Zoomed out view of reads in IGV.

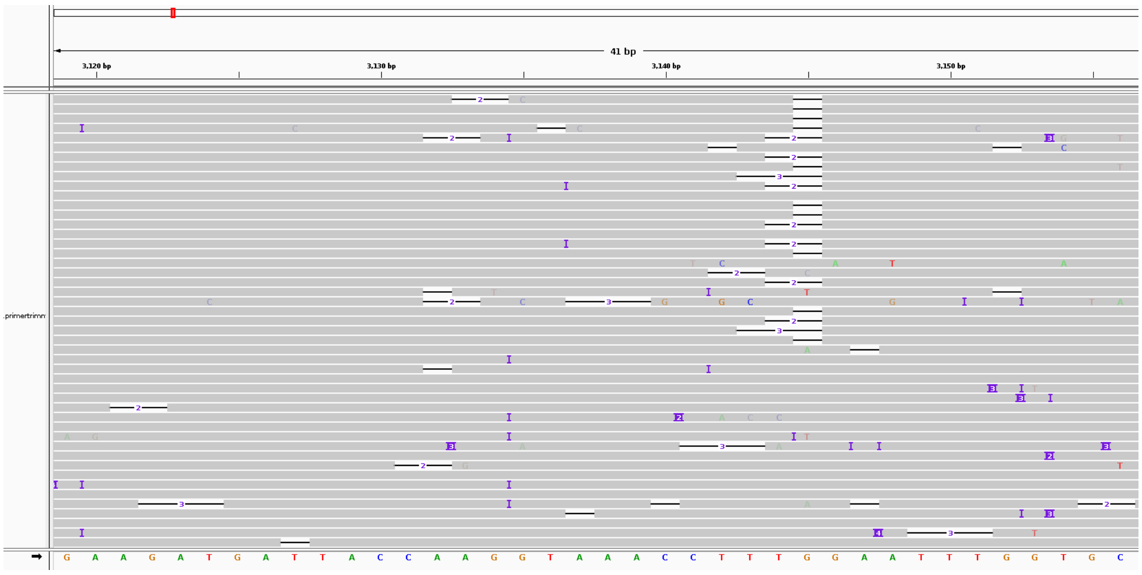
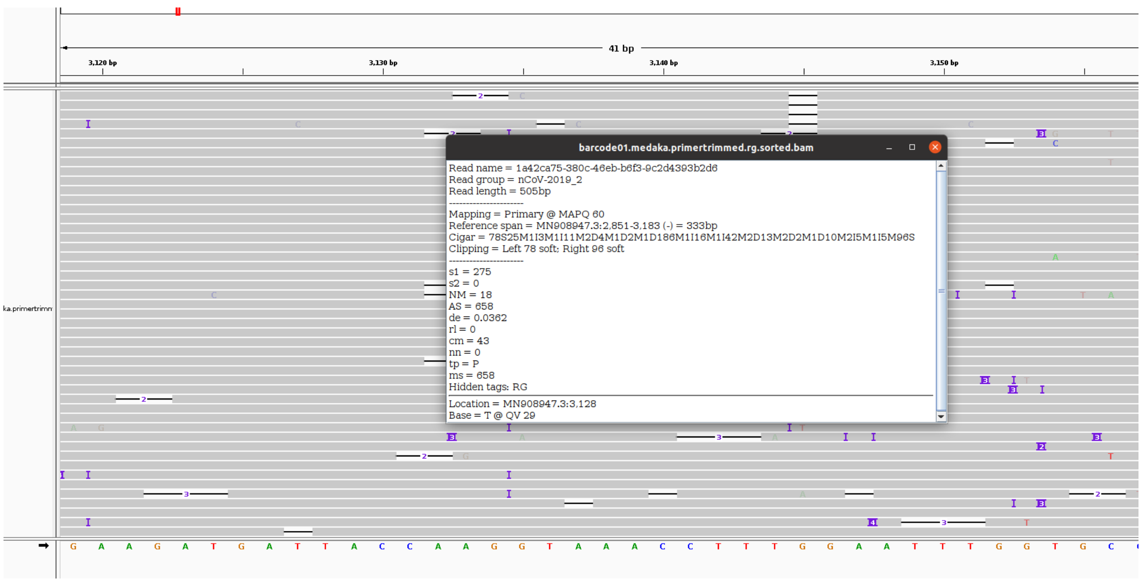
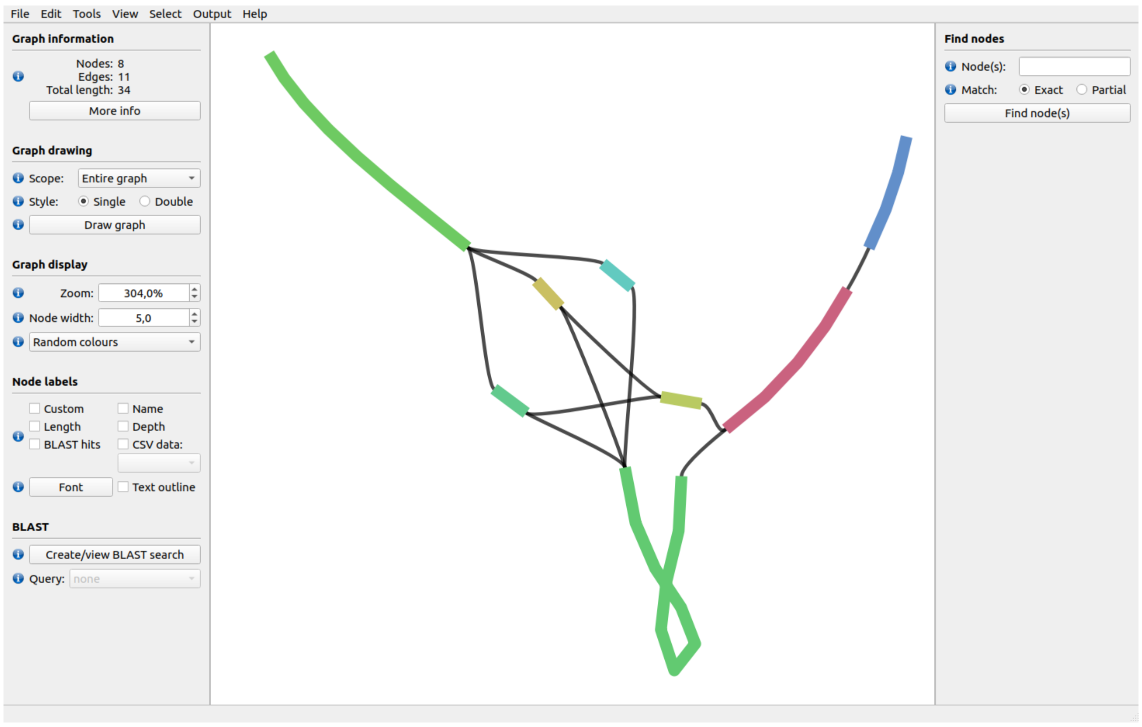

41 bp
3,120 bp
3,130 bp
3,140 bp
3,150 bp
primertrimm
G A A G A T G A T T A C C A A G G T A A A C C T T T G G A A T T T G G T G C

Figure 2: Close up view of reads in IGV.



```
Read name = 1a42ca75-380c-46eb-b6f3-9c2d4393b2d6
Read group = nCoV-2019_2
Read length = 505bp
----------------------
Mapping = Primary @ MAPQ 60
Reference span = MN908947.3:2,851-3,183 (-) = 333bp
Cigar = 78S25M1I3M1I11M2D4M1D2M1D186M1I16M1I42M2D13M2D2M1D10M2I5M1I5M96S
Clipping = Left 78 soft; Right 96 soft
----------------------
s1 = 275
s2 = 0
NM = 18
AS = 658
de = 0.0362
rl = 0
cm = 43
nn = 0
tp = P
ms = 658
Hidden tags: RG
Location = MN908947.3:3,128
Base = T @ QV 29
```

Figure 3: Tooltip of reads in IGV.



Graph information

Nodes: 8
Edges: 11
Total length: 34

Graph drawing

Scope: Entire graph
Style: [✓ Single] Double

Graph display

Zoom: 304,0%
Node width: 5,0
Random colours

Node labels

Custom
Name
Length
Depth
BLAST hits
CSV data:
Text outline

BLAST

Query: none

Find nodes

Node(s):
Match: [✓ Exact] Partial

Figure 4: View of a graph in Bandage.

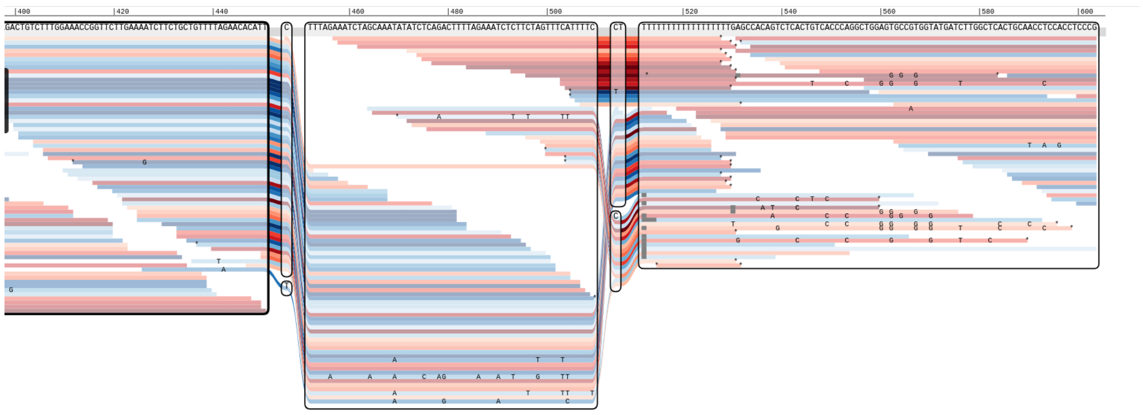


Figure 5: View of SequenceTubeMaps.

1.2 Goals

In our our tool for visualizing the alignment of reads to a pangenome graph, we want to combine feature of Bandage and IGV. We want to make it easily accessible as a web application. Further, we want to test the performance in terms of speed and the maximum number of objects that can be displayed. Our application should support the analysis of pangenomic data and allow users to gain deeper insights into their data. We want to give users with different types of data multiple options to examine it, by providing options to customize the display of the data. Further, our application should be intuitive to use and allow future developers to contribute by implementing additional features. Finally, we give an overview of the resulting features and provide benchmarks for the layout algorithms available to the users Section 4.

2 Preliminaries

2.1 Definitions

In this section, we give definitions of the terms and concepts that are needed to understand this thesis. We define graph structures and biological concepts that relate to these graph structures.

2.1.1 Biological Preliminaries

The molecules that carry the genetic information of an organism are called DNA. In simplified terms, DNA consists of a sequence of bases. Specifically, these bases are called Adenine, Thymine, Cytosine and Guanine. We represent DNA as strings of bases and usually abbreviate the base names using their first capital letter. We call such a string of bases a DNA sequence. The reverse complement of a DNA sequence s is a DNA sequence of the same length but with all of the bases switched with their complementary base (A$\leftrightarrow$T, C$\leftrightarrow$G) and a reversed reading direction. We call a set of DNA strings that includes the reverse complements of each string a genome. We define a read as a DNA sequence of any length that is sampled from a genome. A read is usually a substring of one of the DNA sequences in the genome that contains some differences in the form of one or more bases being replaced, deleted (deletion) or added (insertion). An alignment is an arrangement of reads to a genome. For each read, an alignment denotes the reads origin DNA sequence and the way the read aligns base by base to the DNA sequence.

2.1.2 Pangenome Graph

To define Pangenome graphs we follow the definition of Eizenga et al. [12]. A pangenome graph is a graph data structure that encodes multiple similar input sequences. Since redundant information is only saved once, the data structure is smaller in comparison to saving all sequences separately. A pangenome graph consists of nodes that are connected by edges. Each node represents a piece of DNA sequence. The edges encode concatenations of these sequences. Pangenome graphs can contain the reverse complements of DNA sequences in each node. These graphs are called bidirected graphs. The connections between different sequences and complements is then described by different edge types. Pangenome graphs can be used to represent different relationships in genomic data. They can represent multiple sequence alignments in a compact form, or represent multiple reference genomes with defined paths that denote the references which it is build from. Pangenome graphs can be used to represent different relationships in genomic data. They can represent multiple sequence alignments in a compact form, or represent multiple reference genomes with defined paths that denote the references which it is build from. They can also be used to represent graphs that are used for assembly and represent a set of sequenced reads in this case. All these graphs can be encoded in the Graphical Fragment Assembly format described in Section 2.2.1.

2.1.3 Cigar String

A cigar string encodes how a query sequence aligns to a corresponding reference sequence. It is a sequence of operations. Each operation is preceded by a number, which defines how often the operation should be executed. The following table shows the operations, their semantic meaning, and how they are executed.

Operation	Description	Consumes query	Consumes reference
M	alignment match (can be a sequence match or mismatch)	yes	yes
I	insertion to the reference	yes	no
D	deletion from the reference	no	yes
N	skipped region from the reference	no	yes
S	soft clipping (clipped sequences present in SEQ)	yes	no
H	hard clipping (clipped sequences NOT present in SEQ)	no	no
P	padding (silent deletion from padded reference)	no	no
=	sequence match	yes	yes
X	sequence mismatch	yes	yes

Table 1: Excerpt of the cigar string specification as specified in https://samtools.github.io/hts-specs/SAMv1.pdf

2.2 Data

In this section, we present the data that we worked with and define how the data files are structured, and what information they contain.

2.2.1 Graphical Fragment Assembly

Graphical Fragment Assembly, short GFA, is a format used to represent graphs that contain genomic information. The GFA files are denoted with the extension *.gfa*. These graphs can represent assemblies or variations in genomes[1]. The information is stored line based and each line starts with a single character representing the type of the line. The information

[1]*https://github.com/GFA-spec/GFA-spec/blob/master/GFA1.md*

in each line is tab separated. In the following we give an overview on the information contained in the different line types, that we use.

Segment lines, denoted with an S, define nodes of the graph. A segment line should contain a name field, which should be unique as it defines the name of this segment. Further, the sequence contained in this segment is a required field but if the sequence is not provided it can be filled with a "*". Segments may contain optional fields, which should be formatted in the following way: [tag]:[type]:[value], with [tag] being the name of the field, [type] being being a single case-sensitive letter that defines the format of value, and [value] being the information stored in this field.

Lines denoted with an L define a Link, which represent edges in a graph. A link line contains two fields to denote the start and end point of the edge. These are given as the names of segments. For both segments an orientation is defined as we encode bidirected graphs. These fields contain either a + or - to denote the orientation. The overlap between the two segments is shown in the last required field in the form of a cigar string. It can be omitted by giving "*" as a value. As with segment lines, optional fields can be appended after the required fields in the format described above.

Lastly, path lines, marked with a P, denote a path through the graph. This line contains a path name. The path itself is defined in a separate field as a comma separated list of segment names, which have a + or - appended to define their orientation. Following this field we have a field containing a comma separated list of cigar strings, which define how the segments on the path overlap. If this list is provided, it contains one less entry than the list of segment names. This field can again be omitted by giving "*" as a value. In the following, we show an example for a *gfa* file and provide a corresponding graph in Figure 6.

```
S node1 ACTGTTA
S node2 A
S node3 G
S node4 GTACTAA
S node5 C
S node6 TCTA
L node1 + node2 + *
L node1 + node3 + *
L node2 + node4 + *
L node3 + node4 + *
L node4 + node5 + *
L node4 + node6 + *
P path1 node1+,node3+,node4+,node6+ *
```

Listing 1: Example of a gfa file.

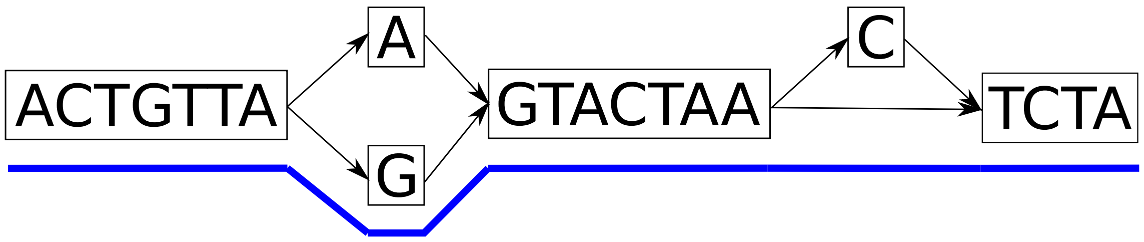


Figure 6: The nodes contain their sequence defined in the segment lines (for this example we kept them unique) and the blue line shows the path that the path line describes. As we do not represent it as a bidirected graph we also omitted the different edge types here.

2.2.2 Graph Alignment Format

The Graph Alignment Format (also Graphical mApping Format), short GAF, is a format that describes sequence to graph alignments. Files in this format have the extension *.gaf*. Each line in this format represents a read that is aligned to a graph defined in GFA format. A detailed definition of all required fields can be found at [2]. We mainly use the field path matching, which defines the nodes that the read spans. Another field that is relevant to us is the optional cigar string field. This cigar string shows how the read aligns to the nodes in the path. In the following, we show an example of such a file with read1 being a read that is the same sequence as path1 in Listing 1 and read2 being a read that covers the

[2] *https://github.com/lh3/gfatools/blob/master/doc/rGFA.md#the-graph-alignment-format-gaf*

```
read1 19 0 19 + >node1>node3>node4>node6 19 0 19 19 19 255
read2 7 0 7 + >node1 7 0 7 7 7 255
```

Listing 2: Example of a gaf file.

whole of node1 from Listing 1. For better readability, we changed the tabs to spaces in this example.

3 Implementation

In this section, we will describe the used tools and libraries that are relevant for the implementation and give an overview of the structure of the application. GIraph is written in JavaScript and thus can be used on any operating system using a browser.

3.1 Tools and Libraries

In this section, we will give a short overview of the most important tools and libraries that we use for our implementation to give some basic understanding of their functionalities. We use the JavaScript framework React[3] to create the website and to manage the state and render cycles of our application. Further, we use Material UI[4] to provide us with ready-to-use React components for buttons, headers, and similar elements. As we want to visualize graphs we use AntV G6[13] for this problem and we draw our AlignmentViewer on a canvas element and use konva[5], specifically react-konva, as a Wrapper for the Canvas API[6]. The version of all used packages is documented in the *package.json* file and the application can be installed locally using npm[7].

3.1.1 React

React is a framework for creating websites and user interfaces that manages state and renders components dependent on that state to the Document Object Model(DOM). As such, developers only need to define the state and create components dependent on that state. In React, variables can be declared as state and React will automatically update the components of our application that depend on these variables. Further, components allow us to split our code in smaller, more manageable parts as each component only needs to know how to render itself. We describe the rendering behavior of a component using the Javascript Syntax Extension(JSX). This syntax extension is similar to HTML in appearance and is converted to actual HTML during the rendering of a component. Also, it follows a tree structure like HTML, meaning a component can have any number of child components, but only one parent component. As such, our application also follows this tree structure. The relation between the components of GIraph is shown in Figure 7.

[3] *https://reactjs.org/*

[4] *https://mui.com/*

[5] *https://konvajs.org/*

[6] *https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API*

[7] *https://www.npmjs.com/*

3.1.2 Material UI

Material UI is a library that provides us with ready-to-use React components for various UI elements. We use it as a base for all of our components except for the GraphViewer and the AlignmentViewer components as these are based on a canvas element created by AntV G6 and konva, respectively. The components come with already applied styling and we use them for most of our UI elements.

3.1.3 AntV G6

AntV G6 is a graph drawing library [13], which we use to render our graphs and it further provides functionalities and various event listeners,which we use to allow the user to interact with the graph. Moreover, AntV G6 allows us to register layout algorithms in addition to the ones that are already provided, and we elaborate on how to add these algorithms in Section 3.2.5. It also allows high customization for the individual elements that are displayed, which we also use in Section 3.2.2 and discuss further possible use cases in Section 5.

3.1.4 Konva

Konva is a Wrapper Library for the Canvas API. Specifically, we use react-konva, which wraps the functionalities even further into React components. For us, this means we can create various canvas shapes as React components. Moreover, this allows us to keep our code structure more uniform and thus more readable as we can create canvas elements as React components, which are directly connected to the state, instead of manipulating the canvas elements manually outside of the React component life-cycle.

3.2 Application Structure

In this section, we will give an overview of the structure of our application and then elaborate on the specific components that we implemented. We give a detailed overview of how these components should be used from the point of view of a user in Section 4. In Figure 7 we see the overarching structure of GIraph. We omitted all components that correspond to regular DOM elements, which we mostly use for the alignment of the other components, Material UI components, that are mostly wrapped in the shown components, and react-konva components. As such the figure only contains components that we created for GIraph.

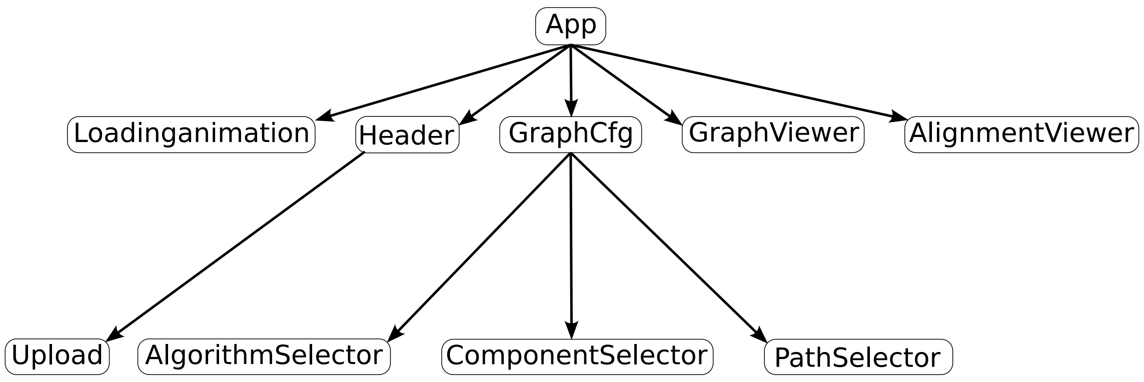


Figure 7: General component structure of GIraph.

3.2.1 Selectors

We have three Selector components, that are all used to set options to customize the graph visualization. All of them are drop down menus and when the user selections an option the graph view is updated. The AlgorithmSelector components allows us to choose a layout algorithm from a fixed list of algorithms. The algorithms are defined in a separate file in the form of an object, which contains labels for the available algorithms. This list of algorithms can be easily expanded by future developers as we describe in further detail in Section 3.2.5.

The ComponentSelector component can be used to choose a specific subgraph, that is a connected component, to be displayed. The list of connected components is sorted by size and each component is assigned a unique identifier, so that multiple connected components of the same size can be distinguished.

The PathSelector component requires that the uploaded graph contains path lines as described in Section 2.2.1. We can choose between one to five paths at the same time, which will then be displayed in the graph by changing the affected nodes to donut shaped nodes. These nodes have a thick outer ring that is split uniformly into section for each selected path that they are contained in. Each selected path is assigned a color from a predefined color palette and the donut ring section of the nodes is colored correspondingly. For the colors we use the colorblind barrier-free color pallet proposed by Masataka Okabe and Kei Ito [8].

3.2.2 GraphViewer

The GraphViewer component handles the visualization of the graph and changes that have to be handled when the users changes the options for the display. These changes include

[8] *https://jfly.uni-koeln.de/color/#pallet*

setting options using the selector components described in Section 3.2.1 and the upload of new files. We use AntV G6 to display the graph. As AntV G6 needs an existing DOM element to append a canvas to, we have to manage it ourselves outside of the React component life cycle. This is also true for any further manipulation of the graph in the form of changes to the display options above. For each of the three selector component, we create an Effect Hook[9] to manage the changes to the graph manually as it is not part of the React life cycle.

3.2.3 Upload

The Upload component consists of a button that opens an input menu on click. We use this component to allow the user to upload a single *gfa* file with and optionally a corresponding *gaf* file. These are then parsed and the corresponding state is updated. Further, we use the library browser-line-reader[10] to read the data in chunks to save memory space during data upload. During the parsing of the files we also compute the connected components of the graph. Then, we reset all other configurations, that are kept in a state, to their starting values.

3.2.4 AlignmentViewer

The AlignmentViewer component uses react-konva to display the alignment at a single selected node on a html canvas element. For this, the data, specifically the *gaf*, file, requires an entry with the "cg" tag which contains a cigar string as described in Section 2.1.3.

We map each character in the cigar string to a corresponding react-konva component, except for Insertions(marked with an "I"). Consecutive insertions are accumulated and displayed as a single number, denoting how many insertions occurred at this position. The reads are sorted left-justified by the first appearing symbol, but ignoring insertions.

As we generate these symbols are react components and store them in an array, each of these components needs a unique key as described in https://reactjs.org/docs/lists-and-keys.html#keys. In our implementation each starting coordinates for the components is only used once, and as such we can use this position as a unique key for the component by converting it to a string. We pass this array to a react-konva Layer component to create a canvas with our elements. As the html canvas element has a maximum size, which depends on the used browser[11], we do not want to create a canvas that is

[9] *https://reactjs.org/docs/hooks-effect.html*

[10] *https://github.com/stanrogo/browser-line-reader*

[11] *https://developer.mozilla.org/en-US/docs/Web/HTML/Element/canvas#maximum_canvas_size*

large enough for all elements. For large alignments this would slow down the website and the canvas might not even support the needed size. Thus, we create a canvas that is the size of the displayed view with some additional space for padding. To allow the user to navigate the whole alignment we implement scrolling using css transform[12] as this provides a fast way of moving the view to the desired position without needing to create a canvas spanning all elements.

3.2.5 Adding Algorithms

We provide the user with a range of layout algorithms to use. We use implementations of AntV G6 and ogdfjs[13], which is based on ogdf[14]. We explain the structure of how the algorithms are stored and how developers can easily expand the available algorithms with implementation of other libraries and their own implementations.

We store the algorithms in an array of objects in a separate JavaScript file. We differentiate between two cases in this array. Firstly, algorithms that are implemented by AntV G6. These consist of an object that only contain their label as defined in the AntV G6 documentation [14]. We store all other algorithms as an object with 2 attributes.The first attribte is a label, which is used in the AlgorithmSelector for the user to see and also in the GraphViewer to register the layout. The second attribute is an object, that should contain a function called "execute()". This function should calculate x and y positions for all nodes. Thus, a new algorithm can be added by expanding the array with such an object. A more detailed overview of how AntV G6 handles custom layouts can be found in the corresponding part of the documentation[15]. In the following, we present an example for a single algorithm of both described types in our implementation.

[12]*https://developer.mozilla.org/en-US/docs/Web/CSS/transform*

[13]*https://github.com/ZJUVAI/ogdf.js*

[14]*https://g6.antv.vision/en/docs/api/graphLayout/guide*

[15]*https://g6.antv.vision/en/docs/api/registerLayout*

```javascript
import G6 from "@antv/g6";
import * as ogdf from "ogdfjs";

export const algorithms = [
  {
    label: "FMMM GorgeousAndEfficient",
    layoutAlgorithm: {
      async execute() {
        let self = this;
        let nodes = self.nodes;
        let edges = self.edges;

        let layout = new ogdf.Layout.FMMMLayout({
          graph: { nodes: nodes, links: edges },
          webworker: true,
          parameters: {
            qualityVersusSpeed: "GorgeousAndEfficient",
            unitEdgeLength: 100,
          },
        });

        const output = await layout.run();
        nodes = output.nodes;
        edges = output.links;
      },
    },
  },
  {
    label: "dagre",
  },
];
```

4 Results

In this section, we present the results of this thesis. The goal of this thesis was to create an application that supports the analysis of genomic data. As such, the results consist of the application and its features. We firstly describe what kind of data is required to run and use our application. Then, we describe how the application should be used. We also provide some benchmarks in relation to graph size and density of edges for each of the different algorithms that is available in GIraph. The code can be accessed at https://git.hhu.de/yumar101/graph-alignment-visualizer and we collect issues and known bugs at https://git.hhu.de/yumar101/graph-alignment-visualizer/-/issues. Further, we provide a Digital Object Identifier[16].

4.1 Using GIraph

In this section, we describe how to display a pangenome graph in GIraph and elaborate how to analyse the graph using GIraphs features. Firstly, a user needs to start GIraph, either by visiting https://franz.cs.hhu.de/graph-alignment-visualizer/ or by cloning the repository and hosting it locally. To access GIraph online, the user has to be connected to the Network of the Heinrich-Heine-University in Düsseldorf. In this section, we will describe how to display a pangenome graph in GIraph. In Figure 8, we show the starting screen of GIraph. The steps described in the following are the same for both methods of accessing GIraph.

[16] *https://doi.org/10.5281/zenodo.7140777*

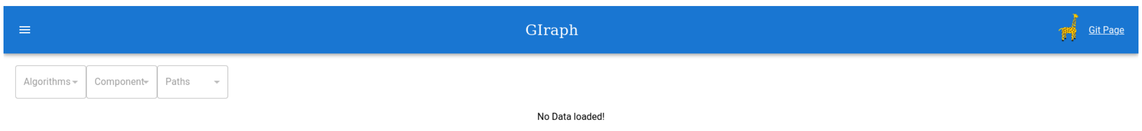


Figure 8: First view when accessing GIraph.

Secondly, we need to upload data. To upload files, we can press the menu button and following that the appeared "Upload" button. The user can see that data was successfully uploaded as the "No Data loaded!" text disappears and an empty canvas with a minimap appears as shown in Figure 9.

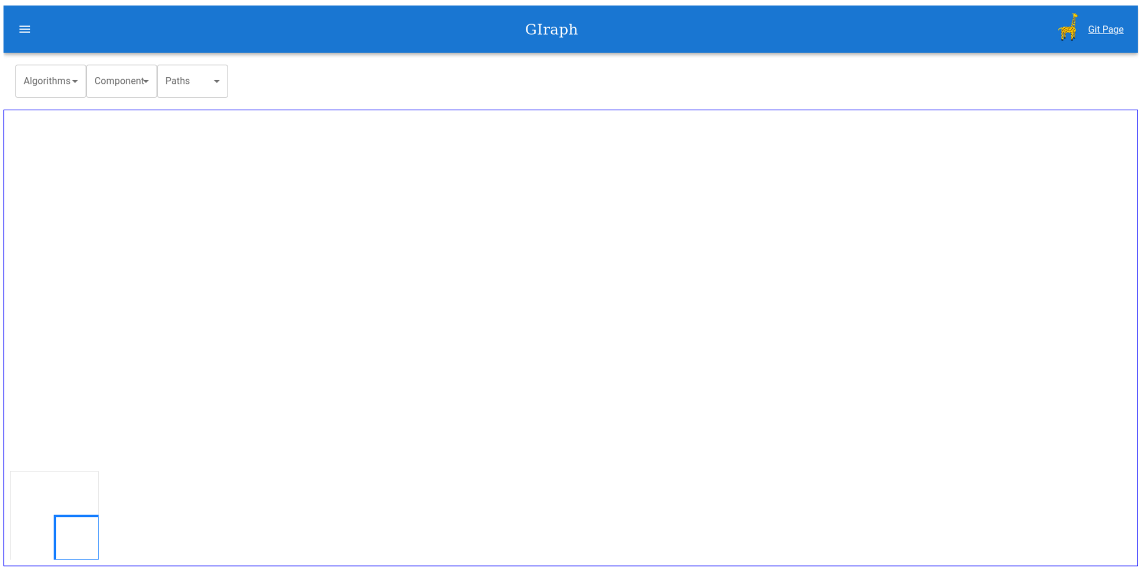


Figure 9: Data was uploaded successfully.

We can upload a *gfa* file and optionally an additional *gaf* file. This enables the selector components, which stops them being grayed out. We can select an algorithm to display our graph from the list of available algorithms.

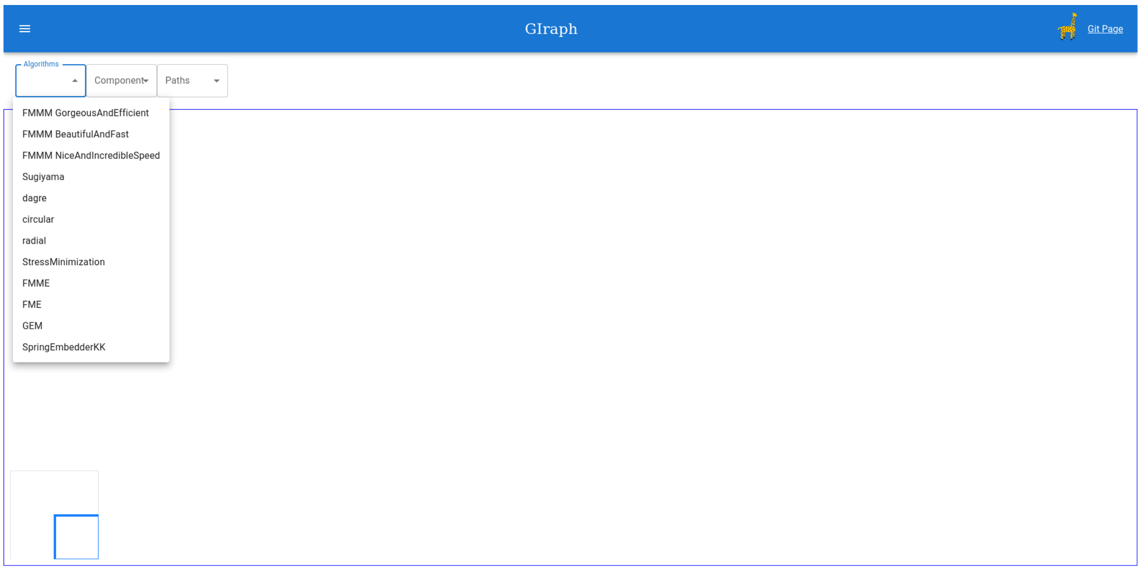


Figure 10: The user can choose an algorithm to layout the data after uploading it.

The implementations of the algorithms are provided by AntV G6 and ogdfjs and documentation regarding these algorithms can be found on the respective websites. After calculating the layout the graph is displayed and the currently visible sector of the graph is marked on the minimap as shown in Figure 11. Alternatively, the user can select a connected component first, then the chosen component is displayed in a grid.

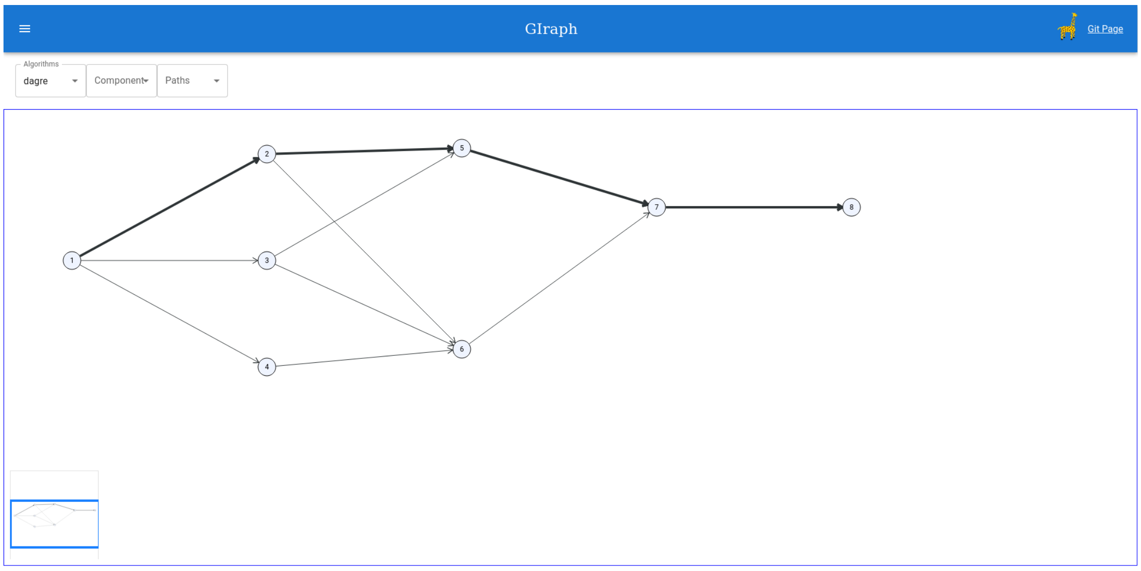


Figure 11: Data rendered with the dagre layouting algorithm.

The minimap is positioned at the bottom left of the graph view and shows a miniature version of the layout. To navigate the graph the view can be dragged with the mouse, by holding the left mouse button, and the level of zoom can be adjusted with the mouse wheel. To increase performance the node labels and edges are not displayed during dragging and zooming of the view. The zoom level can also be set to the maximum level by pressing *ctrl + 1*. Dragging and zooming the graph view is also reflected on the minimap.

The graph view contains additional information about the data, in addition to the connectivity of nodes. Hovering over nodes and edges shows a tooltip with information regarding the number of reads passing this node/edge if a *gaf* file is provided. Moreover, the edge thickness is scaled with the read count as can also be seen in Figure 11. We calculate the read count on a node or an edge by counting how many reads contain this node or are using the edge in question.

The layouts are not guaranteed to prevent visual overlap between nodes or to provide a layout that is visually useful to the user. Thus, the nodes can be moved around to clarify unclear portions of the graph by dragging them while pressing the left mouse button. To increase performance edges adjacent to a dragged node are not visible for the duration of the drag.

Next to the selector element for the algorithm are two further selector elements, for connected components and paths, respectively. These are shown in Figures 12 and 13. The component selector can be used to select connected components of the graph, which causes the graph viewer to only display this specific connected component. These are sorted by size, beginning with the full graph. Each component is assigned a unique ID. The user can see the number of nodes that a component contains in the selector. When a component is selected the graph is rerendered with the currently selected algorithm. When working on large graphs we recommend choosing the relevant component, if applicable, to increase the performance of the graph viewer as there are fewer elements to display.

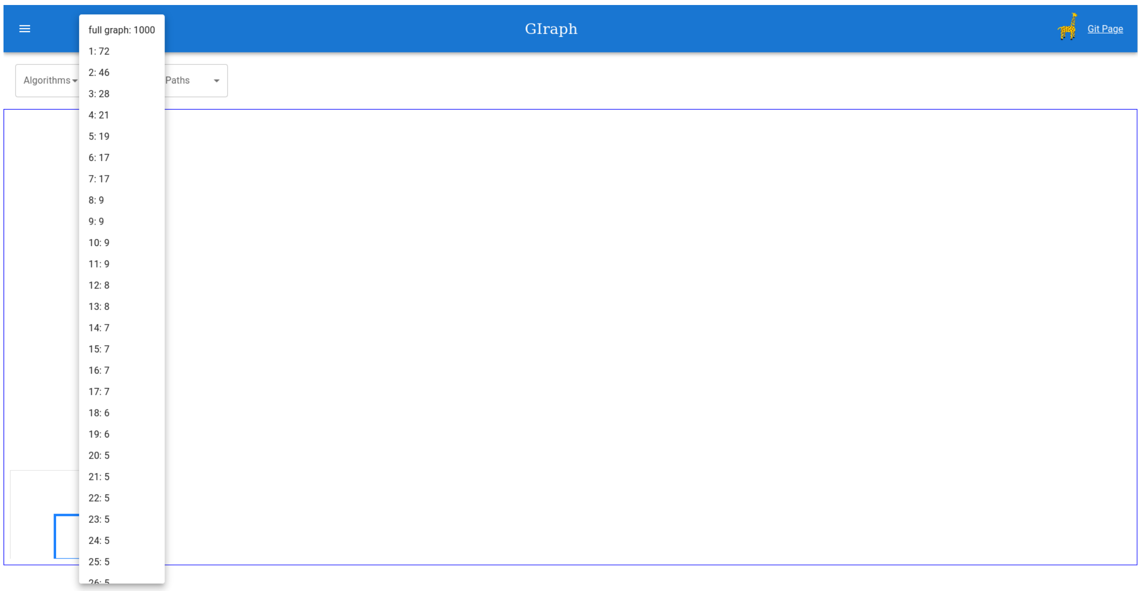


Figure 12: The user can choose a connected component.

The path selector can be used to display paths specified in the *gfa* file. Up to 5 paths can be selected. The affected nodes are then displayed in a donut shape with the ring of the donut colored in equal proportions for each path that they belong to. The path names in the path selector are colored with the same colors, to clarify which nodes belong to which path. The used color palette is colorblind friendly[17]. In Figure 14 we can see a graph with one path marked, and in Figure 15, we can see a graph with two paths marked and the opened path selector with the path names colored accordingly.

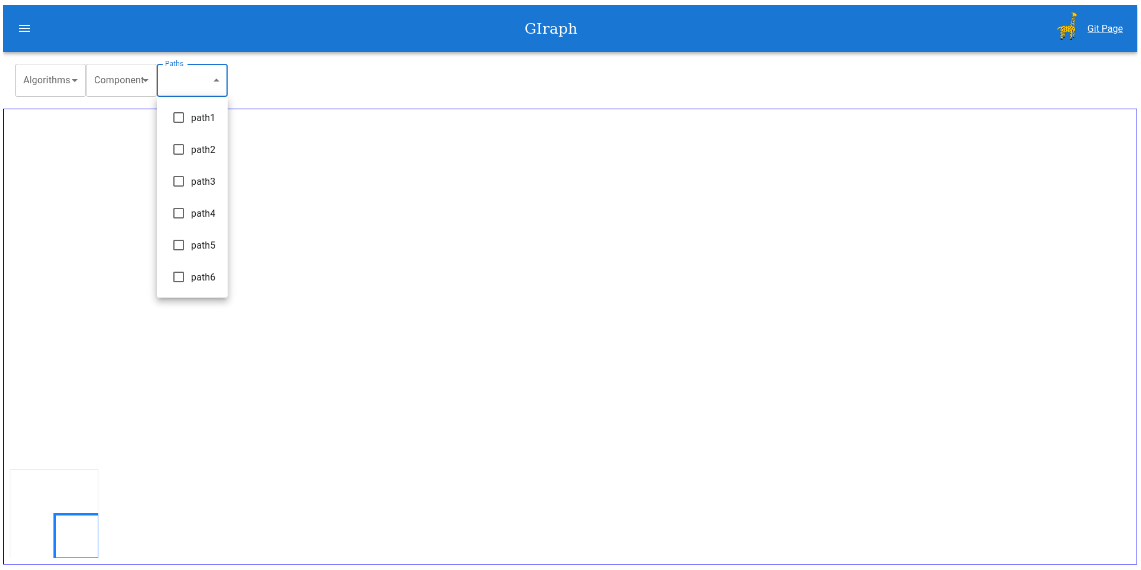


Figure 13: The user can select up to 5 paths.

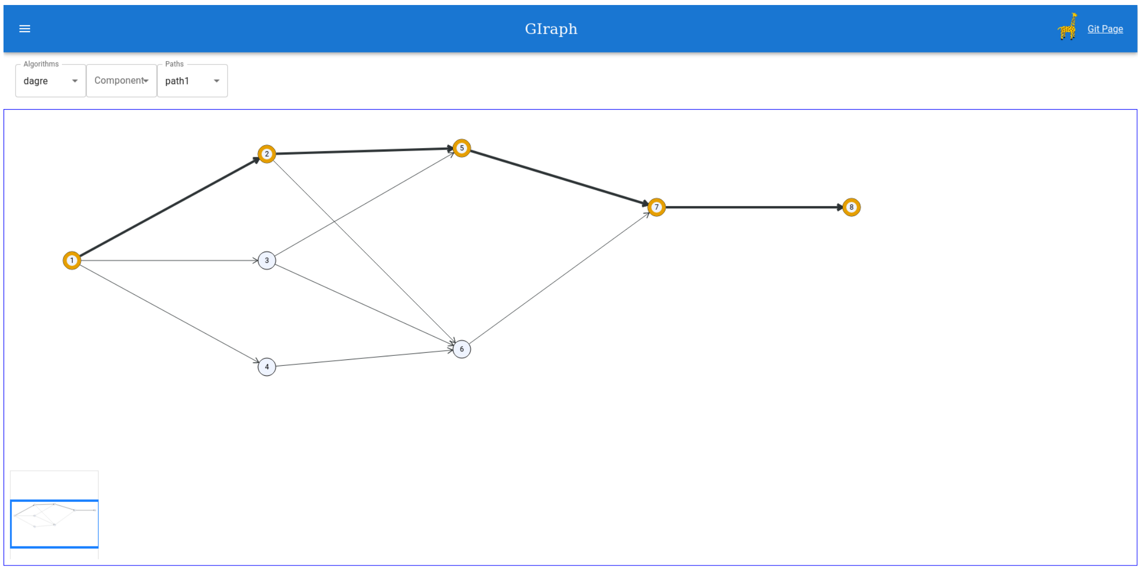


Figure 14: Graph with one selected path.

[17] *https://jfly.uni-koeln.de/color/#pallet*

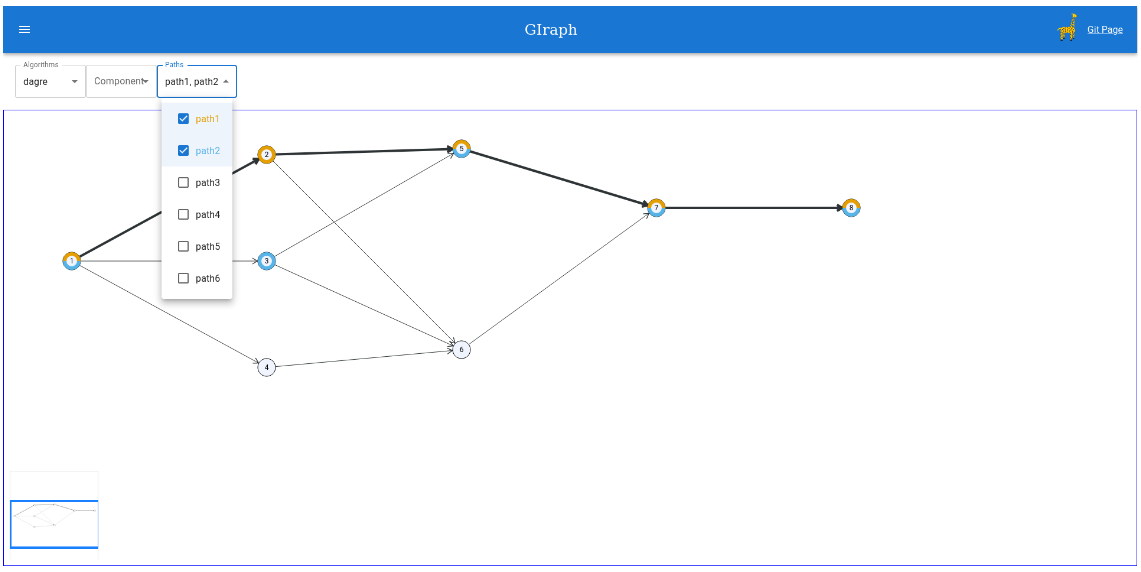


Figure 15: Graph with two selected paths.

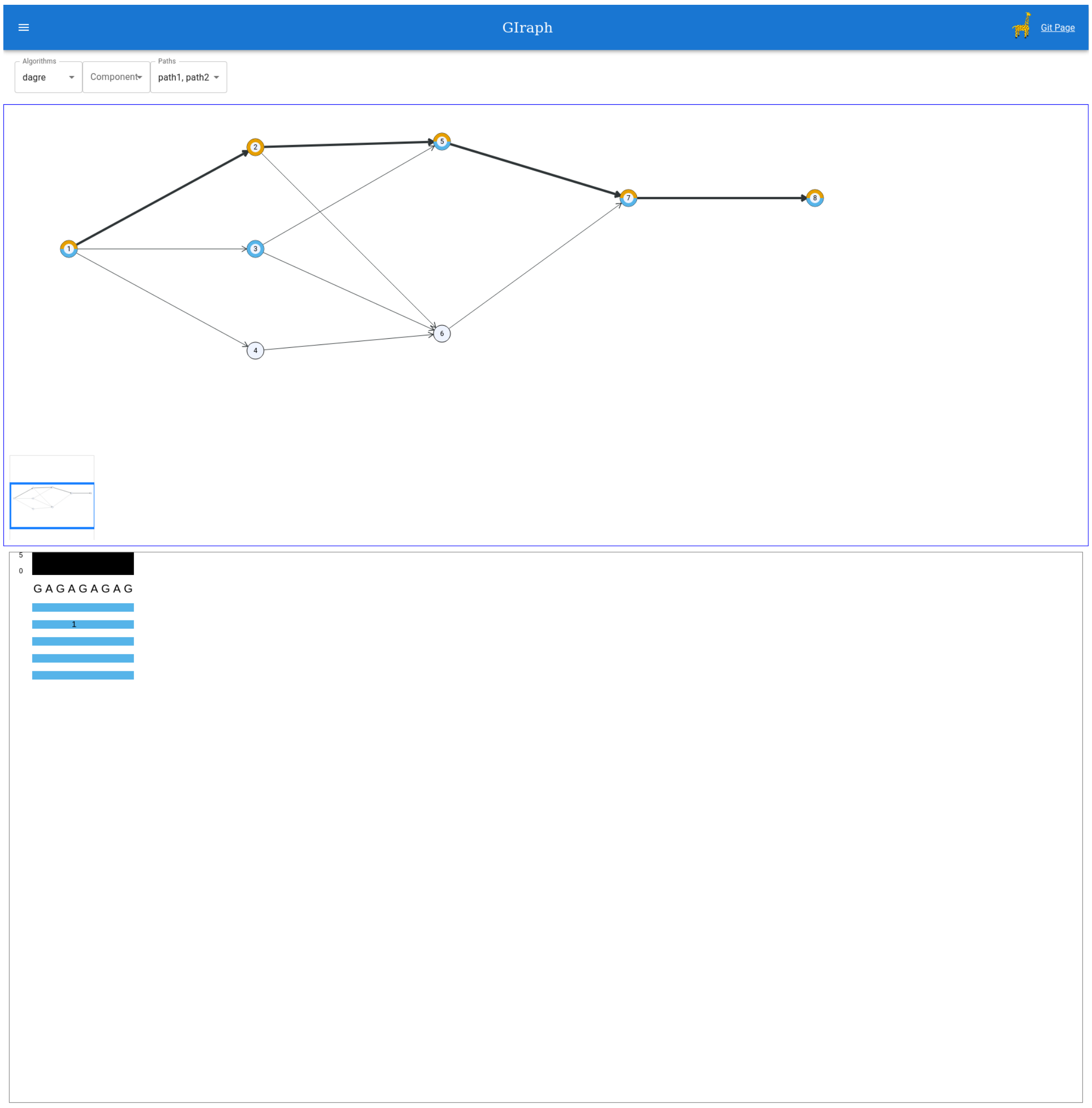


Figure 16: Graph after selecting a node to see the alignment viewer.

Clicking a node opens the alignment viewer below the graph view as shown in Figure 16. The alignment viewer displays the sequence belonging to the clicked node and if a *gaf* that has "cg" tags is provided and there are reads that pass over this node, the reads are displayed below the sequence. In this case, we also show a coverage track above the sequence. The scaling of the coverage track is defined on the left end of the coverage track and starts with zero and shows the maximum value for the coverage on this node. Below the sequence, the individual reads are displayed. These are sorted left justified, meaning reads that start at the beginning are displayed at the top of the alignment viewer. The cigar strings, which encode how the read aligns to the nodes, are parsed to different symbols. Matches are displayed with a blue rectangle, deletions are displayed with a line and insertions are displayed as a number. These symbols align under the corresponding sequence symbol, except for insertions, which don't have a corresponding symbol in the sequence. Rather, insertions are displayed on the position before they occur. Other symbols are displayed right below their corresponding position in the sequence and are differentiated into 2 types, alignment matches and deletions. They are differentiated by different colors and deletions are thinner, to make them easier recognizable as a difference compared to the sequence. The colors for this visualization are chosen from the color palette described in Section 3.2.1. The alignment view has separate scroll bars for vertical and horizontal scrolling, that can also be operated by mouse wheel and shift + mouse wheel, respectively.

4.2 Benchmarks

In this section, we provide some benchmarks for the different layout algorithms in terms of the size and edge density of the graphs. The graphs are generated using a method by Batagelj and Brandes[15] implemented by networkx[18]. As these graphs do not have an inherit structure we also created a ladder graph dataset. It only takes the amount of nodes as parameter. In Figure 17 we see a ladder graph. The overall structure of the graph is shown on the minimap.

[18] *https://networkx.org/documentation/stable/reference/generated/networkx.generators.random_graphs.fast_gnp_random_graph.html*

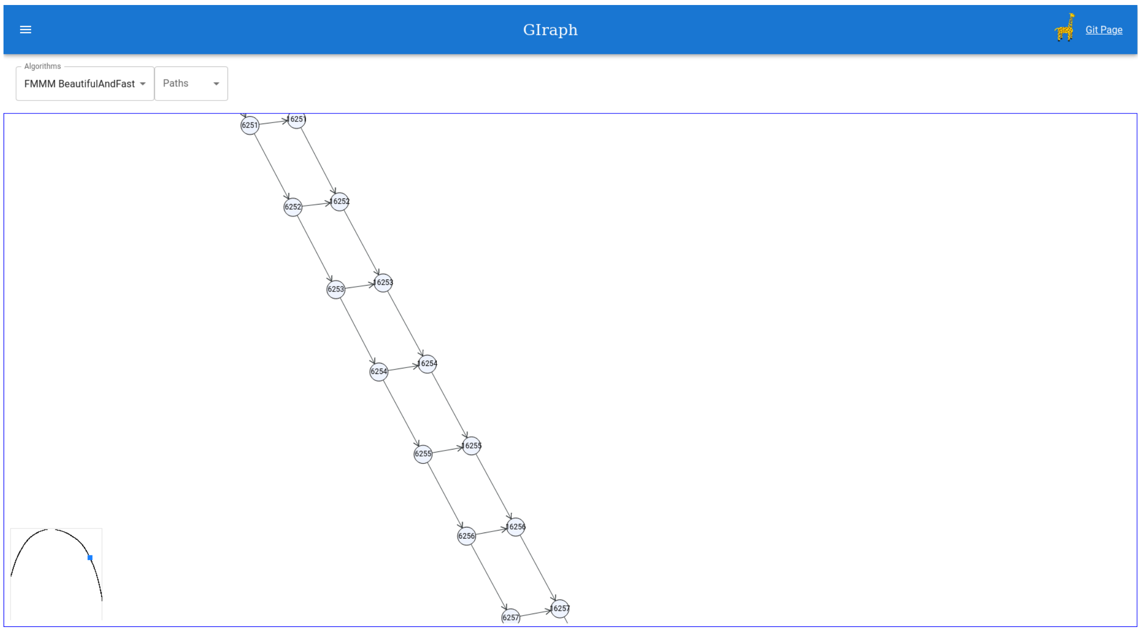


Figure 17: Ladder graph rendered with FMMM.

We provide the python script used to generate the data at https://git.hhu.de/yumar101/graph-alignment-visualizer/-/blob/main/data_generator.py, which also contains the used seeds. The benchmarks are performed on a machine running Ubuntu 20.04 with 8GB of memory and a Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz. We used Firefox 105.0 for the experiments and we restarted the browser between each run to ensure that the conditions for the experiments are as similar as possible. We run each data set 3 times and then average over all runs of the same size and edge density. For the random graphs we need to calculate a value p, which denotes the probability with which an edge exists. As we want to keep the edge density about the same between datasets we calculate p depending on the node count as follows: $p = \frac{nodeCount*avgEdges}{nodeCount^2}$, where avgEdges denotes the average amount of edges that we want per node. For this value we chose 0.5 and 0.7 as parameters as higher values lead to graphs with no visible structure, regardless of which algorithm is used to display it as we end up with a very large connected component with high node degrees and many components that only contain one node. As the ladder graph has a fixed construction based on the node count we denote these entries with "ladder" in Table 2. We set a time limit for the computation of the layout of 3 minutes. Entries are denoted with (T) if they overstepped this limit. If not all of the three runs overstep this limit, we still give a time value but mark it with a (T). The runs that overstep the limit are not used for the calculation of the average. If we run out of memory during the calculation we denote this with (M).

Algorithm	average edge count\node count	10^3	10^4	10^5
FMMM GorgeousAndEfficient	0.5	1.2	6.9	75.5
	0.7	2.1	12.5	-
	ladder	-	22.4	-
FMMM BeautifulAndFast	0.5	1.2	7.9	81.2
	0.7	2.1	11.4	-
	ladder	-	22.6	-
FMMM NiceAndIncredibleSpeed	0.5	1.3	8.0	62.7
	0.7	2.0	11.4	-
	ladder	-	35.0	-
Sugiyama	0.5	1.2	9.0	156.0(T)
	0.7	2.1	72.4	-
	ladder	-	(T)	-
dagre	0.5	1.4	44.2	(T)
	0.7	1.7	57.1	-
	ladder	-	(M)	-
GEM	0.5	2.5	19.7	(T)
	0.7	2.5	16.9	-
	ladder	-	28.1	-
StressMinimization	0.5	4.5	(M)	(M)
	0.7	5.3	(M)	-
	ladder	-	(M)	-

Table 2: Benchmarks of the used algorithms in seconds.

5 Discussion

In this section, we give an outlook on features that are currently lacking and could be implemented in the future.

GIraph allows the user to upload *gfa* files and examine the encoded graph. For an overall structural overview of the graph, it provides features for navigation like a minimap. Further, the user can zoom into and out of the graph to switch between a more detailed view of specific regions and a broad overarching view. The user can navigate the detailed view by dragging it. The sequence information of a node can be accessed by clicking on it. In GIgraph, a multitude of layout algorithms are available. Additionally, multiple encoded paths can be displayed at the same time.

Depending on the graph, none of the available algorithms may yield a layout that satisfies the user. In such a case the user can drag nodes manually to clarify the graph view. We currently have no options to display jumps and walks, which can be defined in more recent GFA versions[19] and also do not consider the input of rGFA files[16].

If a *gaf* file is provided additionally to the *gfa* file, the graph view itself already gives the user more information compared to the above case. The user can see the read flow on the edges as these are scaled according to their read count. This can be used in combination with the colored paths to examine how the reads relate to the different paths. If the reads in the *gaf* file contain cigar strings these can also be displayed by clicking on a node. This function could be improved by allowing the input of an additional *fasta* file to get more detailed information on the reads, for instance not only the positions of insertions but also which bases have been inserted.

The alignment view also has room for improvement. Currently, zooming is only available on the development branch[20] as it is not in a user friendly state.

Further, when loading and layouting large graphs, GIraph can run into issues in terms of runtime and memory, we try to overcome this by allowing the user to choose smaller connected components of the graph, but these might not exist in such a graph. As we can see in Table 2 the algorithms all scale with the size of the data in terms of nodes and edges. Some algorithms are not feasible for large graphs, so we recommend using one of the FMMM variations as they proved to be the most reliable when working with larger datasets. When using a hierarchical algorithm, Sugiyama should be preferred as it is more reliable compared to dagre. If GIraph is used on a computationally more powerful machine some algorithms that run out of memory in our experiments may still successfully calculate a layout, which can be preferable to FMMM. Further, we set a time limit for the experiments, but the user may choose to wait longer to achieve a result.

5.1 Outlook

In this section, we cover possible future features and elaborate on how these can be incorporated in GIraph.

[19] *https://github.com/GFA-spec/GFA-spec/blob/master/GFA1.md#w-walk-line-since-v11*

[20] *https://git.hhu.de/yumar101/graph-alignment-visualizer/-/tree/dev*

5.1.1 Error Feedback

Giraph lacks feedback for the user if something goes wrong. Currently, errors during upload of the data are displayed in the console. If the cause of the error is the content of the file the user does not get feedback on what exactly is wrong with the file. This can be solved by checking each case manually and creating a component that displays errors to the user with a meaningful error message. Further, errors caused by layout algorithms are also only shown in the console and in the case that an algorithm does not throw an error, but also does not calculate a layout, GIraph most likely crashes.

5.1.2 Alignment Viewer

The alignment viewer has room for improvement in multiple aspects. Firstly, tools for linear alignment, like IGV, provide the user with multiple options to filter or color the reads, for example by read direction or mapping quality. Further, additional information about the reads can be displayed. The direction of the read can be displayed by an arrowlike tip at one end of the read signifying the direction. These features can be implemented by defining boolean states for these options, or in case of cut offs input fields that take numerical values, and then using these states to check if a reads satisfies the desired criteria in the calculation of the individual components in the AlignmentViewer component. This can be expanded upon by reading in an additional *fasta* file to get the exact bases that are inserted for each insertion, these can then be displayed in a tooltip when hovering above the corresponding insertion. For this, it is enough to save only the bases that are inserted to save memory, as the remaining information is already in the cigar string. Moreover, tooltips can be used to display information that may only be relevant sporadically, for example the read id to identify specific reads or to show all information about a read at the same time, instead of filtering all reads by the desired criteria. This can be implemented by creating a state and and event handler for each react-konva component that is created for a read, this event handler can then set the state when it is clicked and a tooltip component can be created depending on that state. The tooltip component can be implemented using a Material UI box component[21].

Additionally, the user may want to see the path that an individual read takes through the graph. This can be achieved by again defining a state that holds the information about the chosen read and then displayed by coloring the edges along its path. The coloring of the edges can be achieved similarly to the coloring of the nodes in Section 3.2.2 by adding

[21] *https://mui.com/material-ui/react-box/*

an Effect Hook[22] and can also make use of the color palette used for the paths as we do not utilize all colors of this palette in our visualization.

Another improvement to the alignment viewer is to make some elements of it to always appear in the view. Currently, the sequence of the node that is written above the read tracks disappears when the users scrolls down along the read track, as does the coverage track. As the sequence and coverage information is relevant to all the reads regardless of where they are displayed in the alignment view, they should be kept statically visible above the when the user scrolls down and should only change the view if the user scrolls horizontally.

5.1.3 Graph Viewer

The Graph Viewer has some features that could be imporoved upon or included in future work, which we describe here. Firstly, as seen in the benchmarks the viewer has performance issues with large graph instances in regards to node and edge count. This could be addressed by using a webGL based renderer instead of the current canvas based one. As webGL based rendering is harder to implement, they usually provide less features than canvas based renderers but are more perfomant as they can run some of their code on the GPU[23]. This can be implemented in addition to the canvas rendering and provided to the user as an option for large graphs. For this a component can be created that implements the graph view using a webGL based library, that would probably have less features available, and then based on the users choice either the canvas or the webGL based component is used to render the graph. A potential library for this is NetV.js [17], for which they also provide benchmarks in terms of amount of rendered nodes. There is also such a feature planned for AntV G6, which is currently in development[24].

Another feature that is interesting for the graph view is the display of coverage on the graph. This information is currently only displayed for edges in the form of edge thickness and for the nodes, this information is only available in the tooltips. An approach for this is to scale the nodes size similarly to how it is done with the edges, this leads to issues with layouting, as many algorithms do not take into account differing node sizes. Additionally, the node size might not be easily comparable from a visual point of view. We suggest customizing the node shape to add a "coverage meter" above the nodes that displays the amount of reads passing this nodes passing this node compared to the

[22] *https://reactjs.org/docs/hooks-effect.html*

[23] *https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Tutorial/Getting_started_with_WebGL*

[24] *https://github.com/antvis/G6/pull/3566#issuecomment-1065023635*

node with the highest amount of reads passing through it. The shape could be similar to a loading bar. This can be implemented by defining a custom shape, similarly to how we define custom algorithms in AntV G6 in Section 3.2.5. The corresponding function here is called registerNode and its usage is detailed in the AntV G6 documentation and elaborated upon by some examples[25]. The user should be able to toggle this coverage meter. Additionally, it makes sense to allow the user to select a subset of nodes for which this coverage meter is calculated. If the nodes in question have a lower coverage then the rest of the graph, calculating this coverage meter only for this subset allows the user to see nuanced coverage difference in more detail. For this, the user needs a practicable way to select multiple nodes. AntV G6 provides two already build in functionalities that can be used to select larger areas of the graph at once. One is the brush selection[26], which allows the user to draw a rectangle around an area to select all nodes within. The other on is the lasso selection[27] , which allows the user to draw a line to select all encompassed nodes. For nodes, that are not in the vicinity of each other or if only very specific nodes should be select, these could be selected manually by clicking on them.

Another area of improvement is the layouting. The user may want to layout parts of the graph using different methods. This would be preferred in the case of a graph that in the case of a graph that has a linear structure for the most part and a part that cannot be layed out nicely with methods like Sugiyama [18] or dagre [28]. For this, the node selection methods described in the paragraph above could be used as well and then the user can choose a specific layout method for the selected nodes. The AntV G6 documentation also provides an example[29].

Another feature that can be added is the the display of reads from the *gaf* file, that contain edges that are not contained in the *gfa* file. This can happen during assemblies, if edges are discarded during refinement steps. Such edges can be saved separately during the upload of the data and then displayed in the graph viewer by toggling them on or off. This can be implemented similarly to how the switching between components described in Section 3.2.2 is implemented and can also be visibly marked as edges that are not contained in the original graph.

Furthermore, the navigation lacks an option to automatically search for specific nodes in the graph. For such a feature an additional field can be created where the user can input

[25] *https://g6.antv.vision/en/docs/manual/middle/elements/nodes/custom-node*

[26] *https://g6.antv.vision/en/docs/manual/middle/states/defaultBehavior#brush-select*

[27] *https://g6.antv.vision/en/docs/manual/middle/states/defaultBehavior#lasso-select*

[28] *https://github.com/dagrejs/dagre*

[29] *https://g6.antv.vision/en/examples/net/layoutMechanism#subgraphLayout*

the name of the node that they want to find. For this field the Autocomplete[30] component provided by Material UI can be useful, to provide auto-completion for the node names, which this field can then save to a state. The graph viewer can then use this state similarly to how the paths are colored, but instead of coloring existing nodes, we can add edges between nodes. Finally, the user may want to save the graph after they displayed it with a specific layout and moved nodes around. This data could then be uploaded again to get the same view with the nodes set at the positions from when it was saved. This can be done by creating a component, that creates a custom file format, that contains all information of the graph view, including the positions of the nodes and then lets the user download this file.

[30] *https://mui.com/material-ui/react-autocomplete/*

6 Conclusion

We created GIraph, a web application, to analyze pangenome graphs. We combine features of different tools, for example the visualization of pangenome graphs from Bandage and the visualization of alignments from IGV. This allows users to analyze their graphs using features that they are familiar with and that have been proven to be useful for such analyses. A user can upload graphs and display them with various layouts, depending on the resources that their machine provides. This gives the user an overview of the structure of their graph and shows them where complex regions of the graph are. Further, we allow the user to incorporate read data into the visualization. The read data then alters the graph to show the user where the reads align to the graph. This allows further examination of the alignment on specific nodes. The graph view is interactive, such that the user can focus on specific elements of the graph and move elements around for a more appealing visualization. We give benchmarks for the layout algorithms used in GIraph. These provide a guideline on which algorithms can be used on what graph sizes and what runtimes are to be expected. The FMMM algorithm is currently the most reliable algorithm and shows a scaling behavior that is feasible for larger graphs. We give an outlook on features that could be included in future releases. Examples are filtering options for the graph and alignment viewer, and expanded navigation options to find specific nodes in the graph. Moreover, we give details on how these features could be implemented.

In conclusion, we built a tool to visualize pangenome graphs and corresponding alignments. Although it could be improved upon in further work, GIraph can handle practically relevant datasets. Our tool provides a new combination of visual features, which could help the comprehension of genomic data in future projects.

7 References

[1] Débora YC Brandt et al. "Mapping bias overestimates reference allele frequencies at the HLA genes in the 1000 genomes project phase I data". In: *G3: Genes, Genomes, Genetics* 5.5 (2015), pp. 931–941.

[2] Torsten Günther and Carl Nettelblad. "The presence and impact of reference bias on population genomic studies of prehistoric human populations". In: *PLoS genetics* 15.7 (2019), e1008302.

[3] Mazdak Salavati et al. "Elimination of reference mapping bias reveals robust immune related allele-specific expression in crossbred sheep". In: *Frontiers in genetics* (2019), p. 863.

[4] Stephen T Sherry et al. "dbSNP: the NCBI database of genetic variation". In: *Nucleic acids research* 29.1 (2001), pp. 308–311.

[5] Erik Garrison et al. "Variation graph toolkit improves read mapping by representing genetic variation in the reference". In: *Nature biotechnology* 36.9 (2018), pp. 875–879.

[6] Mikko Rautiainen and Tobias Marschall. "GraphAligner: rapid and versatile sequence-to-graph alignment". In: *Genome biology* 21.1 (2020), pp. 1–28.

[7] Helga Thorvaldsdóttir, James T Robinson, and Jill P Mesirov. "Integrative Genomics Viewer (IGV): high-performance genomics data visualization and exploration". In: *Briefings in bioinformatics* 14.2 (2013), pp. 178–192.

[8] Ryan R Wick et al. "Bandage: interactive visualization of de novo genome assemblies". In: *Bioinformatics* 31.20 (2015), pp. 3350–3352.

[9] Wolfgang Beyer et al. "Sequence tube maps: making graph genomes intuitive to commuters". In: *Bioinformatics* 35.24 (2019), p. 5318.

[10] Stefan Hachul and Michael Jünger. "Large-graph layout with the fast multipole multilevel method". In: (2005).

[11] Alexander Wolff. "Drawing subway maps: A survey". In: *Informatik-Forschung und Entwicklung* 22.1 (2007), pp. 23–44.

[12] Jordan M Eizenga et al. "Pangenome graphs". In: *Annual review of genomics and human genetics* 21 (2020), p. 139.

[13] Yanyan Wang et al. "G6: A web-based library for graph visualization". In: *Visual Informatics* 5.4 (2021), pp. 49–55.

[14] Markus Chimani et al. "The Open Graph Drawing Framework (OGDF)." In: *Handbook of graph drawing and visualization* 2011 (2013), pp. 543–569.

[15] Vladimir Batagelj and Ulrik Brandes. "Efficient generation of large random networks". In: *Physical Review E* 71.3 (2005), p. 036113.

[16] Heng Li, Xiaowen Feng, and Chong Chu. "The design and construction of reference pangenome graphs with minigraph". In: *Genome biology* 21.1 (2020), pp. 1–19.

[17] Dongming Han et al. "Netv. js: A web-based library for high-efficiency visualization of large-scale graphs and networks". In: *Visual Informatics* 5.1 (2021), pp. 61–66.

[18] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. "Methods for visual understanding of hierarchical system structures". In: *IEEE Transactions on Systems, Man, and Cybernetics* 11.2 (1981), pp. 109–125.