Visualizing Differences Between Brain Networks

Rick Hendricksen

May 18, 2015
Abstract

Nowadays, it is possible to obtain brain networks from data measured by a Magnetic Resonance Imaging (MRI) scanner. This opens up new ways of understanding the human brain. One new possibility is comparing two brain networks, which may aid in understanding brain diseases like Alzheimer’s disease. We present an application designed to compare two such networks, either obtained from two people, the same person using different techniques, or the same person at a different point in time.

Visualizing two brain networks is challenging because these networks are already difficult to understand on their own. The usual way to compare two large networks is by computing graph properties, such as the small-worldness. Such graph properties are only a coarse representation of a network, and many interesting properties are lost. Our application shows more information, allowing an expert to not only identify differences and their anatomical locations, but also to investigate their causes and their effects on the rest of the network.
Acknowledgements

I would like to thank my primary supervisor, dr.ir. Huub van de Wetering, for his guidance. His ability and willingness to answer my numerous questions were invaluable. I would also like to thank my second supervisor, dr. Anna Vilanova, for being my window into the medical world. Furthermore, without her there would be many incorrect statements about medical details in this thesis. Dr. Andrei Jalba also deserves my thanks, mainly for his help with some of the intricacies of OpenGL. Finally, I would like to thank ir. Kasper Dinkla for his expertise on matrix visualization, and ir. Paul van de Corput, for his help with structuring my thesis.
Contents

1 Introduction
  1.1 Data .......................................................... 10
  1.2 Glossary ...................................................... 10
  1.3 Previous Work ................................................ 12
    1.3.1 Anatomical .............................................. 12
    1.3.2 Connectivity Focused .................................. 13
    1.3.3 Visual Comparison Methods ......................... 14
  1.4 Task Analysis ................................................ 15

2 Solutions
  2.1 Matrix Visualization ......................................... 17
  2.2 Matrix Ordering .............................................. 21
  2.3 Icicle Tree ..................................................... 23
  2.4 Edge Bundling ................................................ 23
  2.5 Clustering ...................................................... 26
  2.6 Pre-selection ................................................... 28
  2.7 3D Anatomical View .......................................... 30
  2.8 3D Edges ....................................................... 31
  2.9 Artificial Datasets ........................................... 36
    2.9.1 Distance Based ......................................... 36
    2.9.2 Random .................................................. 37

3 Implementation
  3.1 Matrix Rendering ............................................. 40
  3.2 Edge Rendering ................................................. 42
  3.3 Interaction .................................................... 44

4 Results & Discussion
  4.1 Results ....................................................... 45
  4.2 Evaluation .................................................... 46
    4.2.1 Task Analysis ........................................... 47
    4.2.2 Tasks .................................................... 47
    4.2.3 Open Questions ......................................... 48
  4.3 Scalability .................................................... 50

5 Conclusions
  5.1 Future Work .................................................. 51

6 References ....................................................... 53

Appendices
  A Cocomac File Format .......................................... 55
  B Barcelona File Format ......................................... 56
  C Maastricht File Format ....................................... 58
  D XHGFZ File Format ............................................ 60
  E Questionnaire .................................................. 63
1. Introduction

Neuroscientists look into reconstructing human brain networks. A human brain network is a set of regions and a set of connections that connect them. There are two different types of brain networks, both obtained differently. Brain networks can express either functional connectivity or structural connectivity. Functional connectivity is measured by letting the subject perform tasks while in an MRI scanner. When performing such tasks, several regions in the brain will activate, and these regions will become connected in the resulting brain network. An alternative method for obtaining brain networks, measuring structural connectivity, works by tracking fibers in the white matter of the brain.

The surface of the brain, where most of the processing happens, is called the cerebral cortex. Below the cerebral cortex, the brain consists of white matter which contains fibers that connect areas on the cortex. To generate a structural network from MRI data as described by Bastiani et al. [BSGR12], the cerebral cortex first needs to be partitioned into regions. This is called a parcellation, see figure 1.1. Next, the fibers in the white matter need to be converted to connections between regions. Every region will have many outgoing fibers, so the connectivity from one region to another is defined by the number of fibers between these two regions. Generating a functional network requires the use of a parcellation as well, but connections are based on which regions activate together when doing certain tasks, instead of being based on measurements of the white matter.

As mentioned, there are two methods capable of obtaining a brain network using an MRI scanner. Both of these methods have many parameters that influence the resulting network. Simply using a number of possible parameter settings for both methods results in many brain networks with the same regions, because they use the same parcellation. For better tuning of the used parameters it would be useful to compare the resulting networks with each other. Showing the difference between functional and structural connectivity additionally allows neuroscientists to better understand the brain.

Comparing two brain networks can also offer insight into brain diseases such as Alzheimer’s disease. Stam et al. [SJN+07] showed that Alzheimer’s disease affects the brain network, but most details remain unknown. The research by Stam does show that comparing one person’s brain network during different phases of the disease can provide a wealth of information regarding the disease. Bastiani et al. [BSGR12] looked into ways to understand and compare brain networks they generated, and settled on comparing graph properties, such as average path length and small-worldness. The main advantage of this method is that brain networks are very complicated, and reducing them to a single number makes them more comprehensible, and easier to compare. The main disadvantage is that these methods reduce very complex information to a few numbers, inevitably discarding a lot of information.

The parcellation that is used to generate a network can have a hierarchical structure. In essence, there would be several parcellations, where the first splits the cortex up into for example the left and right hemisphere. The second would split each hemisphere up into a number of regions, and a
third would split each region up into more regions. The advantage of this method is that networks generated with less detailed parcellations will be smaller and easier to comprehend. After finding interesting areas, the user can then dive down towards a more detailed network, based on a more detailed parcellation.

This thesis will focus on a way to visually compare two brain networks, but in more detail than the comparison between a few graph properties. We will tackle this problem through visualization, and will attempt to facilitate understanding of the network itself, alongside understanding the differences between them. Methods that focus on network visualization itself are presented, as well as methods that adhere to the anatomical location of nodes on the surface of the brain. All implemented visualizations are compatible with hierarchical information, and are tailored to the needs of neuroscientists, while presenting as much information as possible with a user friendly and fast interface.

1.1. Data

To acquire a brain network, the data which the MRI scanner outputs needs to be converted. An MRI scanner outputs volume data, which is basically a 3D matrix. This is, however, not a network yet. This volume data can be converted to a network using several different methods, one of them is using fiber tracking [CLC+99]. This method detects the fibers that connect areas of the brain. To get a network, we need a parcellation, a subdivision of the surface into regions. Then, fibers can be assigned to these regions. In the resulting network, the strength of a connection between two regions is defined by the number of fibers that connect the two regions. Figure 1.2 shows the whole path from a person to the resulting network. Furthermore, it shows that the volume data can also be used to obtain a triangulated surface mesh. This figure does not show fiber tracking at all, but groups the fiber tracking process under “connectivity computation”, because fiber tracking is not the only commonly used method to obtain connectivity information.

Two brain networks with a hierarchy as described above, along with an associated surface mesh, are the input to our program. This is also shown in figure 1.2, where the input to our application is shown with a thick border. The brain networks can be seen as graphs, which have nodes that correspond to regions in the brain, and undirected edges, which are weighted and their weights are non-negative. The nodes have some supplementary information, such as associated vertices in the mesh, and a name. Furthermore, nodes have a hierarchy, of which you can see a toy example in figure 1.3. For each level in the hierarchy, an adjacency matrix is available as well. An important property of the hierarchy is that not every node has the same amount of children. This property is also visible in figure 1.3.

We assume that the nodes in both input files match. That is, node $i$ on level $l$ must describe the same area of the brain in both datasets. This is an important limitation, because it means random parcellations such as those used by Bastiani et al. [BSGR12] cannot be compared.

1.2. Glossary

**Connection** An undirected connection between two regions in a brain network, with an associated non-negative weight.

**Region** A part of the cortical surface.

**Brain Network** A collection of regions (see Parcellation) and connections between them.

**Adjacency Matrix** A matrix, where cell $i,j$ indicates the strength of the connection from node $v_i$ to node $v_j$.

**Surface Mesh** A set of triangles that describe a surface.

**Hub** A group of nodes that are highly inter-connected.

**Parcellation** A division of the brain surface into regions. See figure 1.1.
Figure 1.2  The source of our data. The same person is scanned by an MRI scanner at different points in time, which outputs volume data. A connectivity computation is performed on this data, for example using fiber tracking. The result is a number of networks per scan, each dependent on the chosen parameters. In all of these networks, each node corresponds to one region in the parcellation. There can be multiple parcellations that, together, describe a hierarchy. Each additional parcellation can be an input to the connectivity computation, or the connection information at higher levels can be aggregated. Finally, the volume data can also be used to create a mesh of the surface. Each vertex in that mesh can then be assigned to a region in the parcellation, and therefore a node in the network. The four pieces with a thick border become the input to our application.

Figure 1.3  A toy input dataset for our visualization. It contains nodes with a hierarchy and adjacency matrices, as well as a list of vertices and a list of faces. Every face consists of 3 indices, each of which points to a vertex. Each node also has a list of associated vertex indices. The levels in the hierarchy are indexed, the lowest level is level 0, while the highest level in this example (the root) is at level 2.
Figure 1.4  Anatomical brain visualization by Böttger et al. [BSL+14], which uses edge bundling [Hol06] to enhance their results.

Figure 1.5  Anatomical brain network visualization by Foucher et al. [FVC+05].

Fiber A pathway that connects points on the surface of the brain. These fibers are found in the white matter of the brain. The white matter is mostly found below the surface of the brain. See [CLC+99].

Streamline A path through the brain that follows the directions of fibers. Obtaining these streamlines is called deterministic tractography, and is described by Tuch [Tuc02].

1.3. Previous Work

This thesis is about visualizing the difference between two brain networks. Visualizing differences is dependent on visualizations for brain networks in general, for which a summary paper is available by Margulies et al. [MBWG13]. We divide the relevant previous work into two categories, Anatomical and Connectivity focused. The first, as the name suggests, focuses on the anatomical locations of regions, while the second mostly disregards anatomical locations in favor of visualizing the connection information more clearly.

Most of the previous work uses slightly different terminology than we do. They generally use graphs, with nodes and edges, while this thesis mostly uses networks with regions and connections. In our case, a region is simply a node with some additional information, a connection is simply an edge with some additional constraints, and a network is simply a graph with some additional data. During this section we will use graph and network interchangeably, because they are compatible during this section. In later sections we will stick to regions, connections and networks, because those names better describe the data we actually have.

1.3.1. Anatomical

Anatomical methods focus on the actual locations of each node. The main advantage of these methods is that they allow the expert to easily identify which node is which. This does, however, impose a constraint on the position of each node. Each node must be placed in a location that corresponds to their anatomical location. This can make it harder to visualize connections. An example of an anatomical approach to brain network visualization can be seen in figure 1.5.

The solution by Foucher et al. [FVC+05] draws connections around the brain, but an alternative solution is simply rendering connections on top of the brain as described by Xia et al. [XWH13], see figure 1.6. The disadvantage of the method by Xia et al. is that the edges do not really seem to fit together with the 3D mesh, and generate a lot of clutter. A similar option was taken by Böttger et al. [BSL+14], see figure 1.4, but they use the edge bundling method by Holten [Hol06] to improve their results, while also allowing for larger networks.
1.3.2. Connectivity Focused

Connectivity focused methods focus on showing the connections between regions, rather than anatomical locations. Figure 1.7 shows an example of such a visualization. The main advantage of connectional methods in general is that they are not limited by the anatomical locations of each region. This allows for more freedom with regards to positioning and rendering. The main disadvantage is that the coupling between the 3D locations and the nodes in a connectional visualization can be unclear.

Since connectional visualizations disregard anatomical locations, classical graph visualizations can be used as well. An example is the solution by Alper et al [ABHR+13], which describes two methods to visualize the difference between links in two weighted graphs. They describe a matrix based

The method by Xia et al. has issues with clutter, which is likely to become unmanageable when using a 1000-node dataset. Furthermore we believe that rendering edges around the brain, like the solution by Foucher et al., is more effective.
Figure 1.9 A matrix visualization with collapsing hierarchies. [vHSD09] This shows the same graph at two levels in the hierarchy.

Figure 1.10 Methods for visually comparing two graphs. Top: juxtaposition, bottom-left: superposition, bottom-right: explicit encoding.

layout, and a visualization based on a two dimensional node-link diagram. An example of their matrix visualization can be seen in figure 1.8. Note that Alper et al. already focus on visualizing differences between two networks, and we will extend their solutions in this thesis.

An extension to general adjacency matrix visualization was proposed by van Ham et al. [vHSD09]. They describe a method where nodes are collapsed based on their hierarchy. This is especially useful when the graph is very large, and there are insufficient pixels available to render the original matrix. An example of a matrix shown at two different hierarchy levels is shown in figure 1.9.

1.3.3. Visual Comparison Methods

Most of the previously described earlier work focuses on brain network visualization, or more generic graph visualization. However, the focus in this thesis is on visualizing differences between two such networks. Gleicher et al. [GAW+11] describes three generic ways to compare two entities with visualization. These three methods can be seen in figure 1.10.

Juxtaposition Separate entities are shown separately. That is, they can be shown side by side, or the view can alternate between showing the first and second graph. This method is relatively easy to implement but relies on the user’s ability to remember the first graph when looking at the second in order to find differences. The user can be assisted with selections and highlights, which allow the user to select a region of interest and see the same area lit up in the second graph.

Superposition The two entities are rendered in the same space. An example is drawing both graphs on top of each other, or slightly offset. The main issue with this method is that properties which are present in both entities get drawn twice, which causes clutter.

Explicit encoding This option uses some extra information obtained using a preprocessing step. For example, the differences between the two entities are found first, which allows properties which are the same in both networks to be rendered only once, potentially in a different color than the properties that differ. In general, this method is used as an extension to the superposition method.
1.4. Task Analysis

Neuroscientists obtain two brain networks, and want to answer the following questions:

1. To what extent are two regions connected. Does this connection differ between the two networks?

2. How connected is a region? Is there a significant difference between the two networks? Maybe the second network has fewer, but stronger connections?

3. Where are hubs, and do they have the same size for both networks? Are they the same otherwise?

4. Which other region does this region mainly connect to? Is it the same region for both networks?

5. Where, anatomically, are the differences located?

6. Which regions are the most different, and which regions are identical?

Besides answering the questions above, the expert we spoke to had a few specific requirements. Showing anatomical information, question 5, had to be solved using a 3D rendering of the brain. This 3D rendering also needed to contain information on the connections in the network.

During a different expert meeting, it was mentioned that a pre-selection would be desirable. That is, the user should be able to select several regions, and only see the part of the network connected to those regions. This feature would allow an expert to focus on the part he is interested in, without being overwhelmed by the information in the rest of the network. Note that this was not part of the original scope of the project, but was implemented nonetheless.
2. Solutions

Our initial visualization for connectivity information is based on an adjacency matrix visualization, which has been augmented to support visualizing differences. This visualization proved insufficient, so several other, linked, visualizations were added. These visualizations include an edge bundling visualization, which is loosely based on standard node link diagrams. Furthermore, a context-focused 3D visualization was implemented whose main goal is providing the expert with a global idea of where things are. This was needed because the previous visualizations all ignore spatial locations, which makes them unintuitive in some aspects.

The final application consists of three parts, the 3D view, the 2D view, and a control panel, all shown in figure 2.1. The control panel shows information on whatever the mouse hovers over in the 2D or 3D views. The 3D view consists of the solutions described in sections 2.7 and 2.8. The 2D view consists of the solutions described in sections 2.1, 2.3, and 2.4. For each of the described visualizations, interaction is described in section 3.3.

2.1. Matrix Visualization

The most straightforward method for rendering graphs is using a node-link diagram. However, standard node-link visualizations generate a lot of clutter when used with large graphs: many edges tend to be drawn on top of other edges, or even nodes. This means we can no longer see part of the data, which is an issue.

An alternative method is the use of an adjacency matrix. An adjacency matrix is a matrix where cell $i,j$ contains the strength of the connection from node $i$ to node $j$. We will focus on undirected graphs, so that means that the connection from $i$ to $j$ is equal to the connection from $j$ to $i$.

An adjacency matrix inherently never has overlapping cells, which means it never occludes part of the data. Furthermore, every edge needs only one pixel, but at the cost of needing to visualize the complete graph. This is, however, not necessarily an issue, since it always encodes absent edges as well, which can be desirable.
Even though adjacency matrix visualization seems nice, large graphs will still be hard to understand. An attempt to avoid this issue is summary nodes. The idea is that nodes can be grouped based on the hierarchy, and those groups can be seen as nodes again. This can be used to give the user a higher level overview of the network. This whole concept is explained in more detail by van Ham et al. [vHSD09].

Our solution is based on the solution by van Ham et al. [vHSD09], but we extended it to support visualizing differences. Alper et al. [ABHR+13] proposed a way to visualize differences between two adjacency matrices. Their method is based on overlaying the two matrices, which results in two values per cell. One value for each graph. This results in a new problem, which is how to visualize two values in one cell. Figure 2.2 shows several encodings that can be used for this task. The leftmost option simply shows the actual values for each cell. Difference visualization using these encodings can be added on every level independently, so it does not conflict with the idea of summary nodes.

We will define the following scaling functions for numerical values $a$ and $b$. One uses logarithmic scaling, the other does not. Our application can switch between the two functions, but in the following paragraphs we will only use the logarithmic version to avoid unnecessary duplication. Also note that the log base does not affect our scaling function.

$$
\langle a \rangle_b = \frac{\log(a + 1)}{\log(b + 1)}
$$

In the paragraphs below, the encodings in figure 2.2 will be described in more detail. The first four were proposed by Alper et al. [ABHR+13], the last three are new. Note that the values $v_1$ and $v_2$ correspond to the two values for each cell, and the value $H$ corresponds to the highest value of all cells in both networks. All encodings use the HSV color model [Smi78], and it assumes the values for each component $h$, $s$ and $v$ to be between 0 and 1. Each equation mentioned in this section is accompanied by an illustration which shows the result of that equation when sampled at a few discrete points.

Circles The first encoding (figure 2.2a), drawing circles inside each cell, works well for very small matrices. Its main disadvantage is the amount of unused pixels, even in cells with a full-size circle. These unused pixels manifest in the corners of each matrix cell, which is caused by the mismatch of square cells and circles to show values. This means that for large matrices, where each cell is only a few pixels large, only very few pixels remain to visualize the data. This was the first encoding mentioned by Alper et al. [ABHR+13], but is not implemented in our application.
When two circles overlap, the color is set to white, as shown in the example in figure 2.2, and in the illustration for equation 2.2.

**Bar charts**  The second encoding (figure 2.2b) is based on the highly common bar chart. Again, this works nicely for small matrices, but just like the circle encoding it loses a lot of pixels. Furthermore, it introduces horizontal patterns which are not part of the data. Also, it can sometimes be difficult to see which of the two bars is the left one. In other words, cell boundaries can be hard to identify. This encoding is, however, a straightforward solution, you have two values per cell, so you draw two bars. This makes it an intuitive solution, which works well for small matrices.

The equation below shows how bar heights are computed. Note that a value of 1 means the cell is completely filled vertically, while a value of 0 indicates that a bar is completely absent.

\[
\begin{align*}
\text{leftBarHue} &= \text{red} \\
\text{rightBarHue} &= \text{blue} \\
\text{saturation} &= 0.8 \\
\text{leftBarHeight} &= \langle v_1 \rangle_H \\
\text{rightBarHeight} &= \langle v_2 \rangle_H 
\end{align*}
\]  
(2.3)

Note that, in the illustration accompanying the above formula, the first sample in both directions is 0, so the first red bar in every row is of height zero. Similarly, the first blue bar in every column is of height zero.

**Diagonal**  The previous encoding, bar charts, created unwanted patterns in the visualization. Furthermore, it could be difficult to see where the cells were. Is the red bar the leftmost bar, or was it the blue one? The third encoding (figure 2.2c) visualizes the values for each graph in a half of the cell. The main advantage of this encoding is that it is always clear where the cell boundaries are. Each half contains a grayscale representation of the value inside the cell, so the darker the color, the lower the value. This encoding does introduce distracting diagonal lines, which makes it unsuitable for many situations. Colors are computed as follows:

\[
\begin{align*}
\text{hue} &= 0 \\
\text{saturation} &= 0 \\
\text{value1} &= \langle v_1 \rangle_H \\
\text{value2} &= \langle v_2 \rangle_H 
\end{align*}
\]  
(2.4)
Squares The squares encoding, proposed by Alper et al. [ABHR+13] like the previous encodings, is meant to avoid distracting patterns. Like the diagonal encoding, this encoding shows the values for both graphs in a grayscale color. The first graph’s value is shown in the outer square, while the second graph is shown in the inner square. The main advantage of this encoding is that it does not introduce any distracting patterns, while still encoding both values in each cell. Alper et al. [ABHR+13] described these first four encodings as well, and decided that the squares encoding was their best. Colors are computed exactly like they are for the diagonal encoding, see equation 2.5.

\[
\begin{align*}
\text{hue} & = 0 \\
\text{saturation} & = 0 \\
\text{value1} & = \langle v_1 \rangle_H \\
\text{value2} & = \langle v_2 \rangle_H
\end{align*}
\] (2.5)

Diff The previous encodings all focused on the two values in the cell, but maybe the difference between the two values is more interesting. We came up with an encoding to visualize the difference between the two values. This encoding shows only the absolute difference between the two values. The amount of difference is encoded in the value of the color, \(v\) in the HSV color model. The saturation is set to the maximum, and the sign of the difference decides whether the hue is set to red or blue. See equation 2.6. One added bonus to this encoding is that it does not require any geometry. This means it still works when only one pixel per cell is available.

\[
\begin{align*}
\text{hue} & = \begin{cases} \text{red} & \text{if } v_1 > v_2 \\ \text{blue} & \text{otherwise} \end{cases} \\
\text{saturation} & = 1 \\
\text{value} & = \langle |v_1 - v_2| \rangle_H
\end{align*}
\] (2.6)

Diff Squares When working with the previous encoding, we noticed that it was not possible to distinguish between for example \((0.1, 0.1)\) and \((1, 1)\), with \(H = 1\), while these two pairs are very different. To fix this issue, we came up with this new encoding, inspired by the Squares encoding by Alper et al. [ABHR+13]. This encoding shows the average of the two values in the border, and the diff, like in the previous encoding, in the center. The color for the center is computed exactly like in equation 2.6, while the color for the border is computed using the following equation.

\[
\begin{align*}
\text{hue} & = 0 \\
\text{saturation} & = 0 \\
\text{value} & = \langle \frac{v_1 + v_2}{2} \rangle_H
\end{align*}
\] (2.7)

Saturation Diff As we moved to larger datasets, using the previous encoding became an issue. There were insufficient pixels available to properly draw the boxes and their centers. To fix this, we came up with an encoding that does not require additional geometry, while still being able to distinguish between two cells with values such as \((0.1, 0.1)\) and \((1, 1)\), with \(H = 1\).

This encoding (figure 2.2g) is very similar to the diff, but a bit more intricate. The hue is again decided by the sign of the difference between the two values, but the other two values are
Figure 2.3  Tree ordering. The root of the subtree that is being reordered is circled in red. First the children of the root are reordered (left). Then the children of the children of the root (middle). Then the last level, which does not have any children to reorder, so nothing needs to be done. The corresponding adjacency matrix is reordered as well.

different. The saturation is set to the relative difference, and the value is set to the maximum value. Equation 2.8 describes this in more detail.

\[
\text{hue} = \begin{cases} 
\text{red} & \text{if } v_1 > v_2 \\
\text{blue} & \text{otherwise}
\end{cases}
\]

\[
\text{saturation} = \begin{cases} 
1 - \frac{\min(v_1, v_2)}{\max(v_1, v_2)} & \text{if } \max(v_1, v_2) > 0 \\
0 & \text{otherwise}
\end{cases}
\]

\[
\text{value} = \langle \max(v_1, v_2) \rangle_H
\]  

Equation 2.8

The main advantage of this encoding is that it still shows strong links which are the same in both networks. Figure 2.2g shows that this encoding is capable of distinguishing between the bottom two cells. Furthermore, it still displays the upper right cell, but in gray. The same property can be observed in the illustration for equation 2.8, because unlike the diff encoding, all cells have a unique color. Another large advantage of this encoding is the fact that it does not add any geometry, which means it works well at small sizes. Because of its many advantages, this is the default encoding in our application.

2.2. Matrix Ordering

Matrix visualization [WTC08] in general is not effective without a good node ordering. Because of that we came up with a node ordering method capable of handling hierarchies, by extending an existing node ordering method. The main idea of our modification is that nodes with a common parent are sorted amongst themselves, and are kept together. This means it is never possible for two nodes with a common parent to have a node between them with a different parent. A small example can be seen in figure 2.3, where the direct children of the root node are reordered which causes the children of those nodes to move as well, because we effectively reorder subtrees.

When ordering a matrix, you want nodes with a similar neighborhood to be next to each other. See figure 2.4, where node e and node a have a very similar neighborhood, so they should be close together. Figure 2.4 also shows the reordered variant, where node a and node e are next to each other in the matrix. This problem can be described as a minimization problem, where we want to minimize the difference in neighborhood between two adjacent cells. This can be described as a Traveling Salesman Problem [Kru56], using a similarity measure which looks at neighborhoods. A suitable similarity measure for this purpose is the Tanimoto similarity [Tan57], as shown in equation 2.9.

\[
d(A, B) = \frac{A \cdot B}{|A|^2 + |B|^2 - A \cdot B} \quad (2.9)
\]
Figure 2.4 Neighborhoods. Node a and node e have a very similar neighborhood. The reordered matrix is shown on the right. The cost of both orderings is also shown, and is based on the similarity between two columns.

Figure 2.5 A 2-exchange as used in the 2-OPT algorithm.

Figure 2.6 Reordering a subtree, shown in black. Instead of inserting a dummy node to make a cycle, use the successor and predecessor (gray). This is modeled using an unbreakable link (red) to connect the predecessor and successor. The other nodes, in blue, can be reordered to optimize the path.

The Tanimoto similarity takes as input two \( n \)-dimensional vectors \( A \) and \( B \), where the value for each dimension is in the range \([0, 1]\). The Tanimoto index is basically the weighted version of the Jaccard index \([Jac12]\), shown in equation 2.10. The Jaccard index takes as input two sets, and outputs a number closer to 1 the more equal the given sets are. A distance matrix using the Jaccard index has the value \( 1 - J(M[i], M[j]) \) on position \( i,j \), where \( M[i] \) and \( M[j] \) are columns in the original adjacency matrix.

\[
J(A, B) = \frac{|A \cap B|}{|A \cup B|} \quad (2.10)
\]

Now we have a good similarity measure, the distance measure can be obtained using \( 1 - S(a, b) \), where \( S(a, b) \) is the similarity between node \( a \) and node \( b \). With the distance measure, we can start solving the TSP. We used an algorithm called 2-Opt \([Cro58]\), which is a local search algorithm that starts with an initial tour, and then iteratively improves the result by using so-called 2-exchanges. In a 2-exchange the idea is that the pair of edges \( e_1 = \{u,v\} \) and \( e_2 = \{x,y\} \) is swapped with another pair of edges \( f_1 = \{u,y\} \) and \( f_2 = \{x,v\} \), to get a shorter tour. Such a two exchange is shown in figure 2.5. This algorithm is easily implemented, and widely used.

After solving the TSP, we end up with a cycle. However, a node ordering is a path, not a cycle. The trick here is to insert a dummy node, before solving the TSP, whose distance to every non dummy node is zero. The optimal solution will put the dummy node between two nodes that are far apart, which means the dummy node is the ideal location to cut the cycle open and create the path we were looking for. This method is described in more detail by Climer and Zhang \([CZ06]\). Inserting a dummy node is only needed when sorting the whole matrix. When sorting a subtree,
the neighbors of the leaves in the subtree are treated as fixed, so we are actually looking for an ideal path that starts with the left neighbor \( a_1 \) and goes to the right neighbor \( a_2 \), while visiting all leaves in the subtree that is being reordered. Figure 2.6 shows the nodes \( a_1 \) and \( a_2 \) in gray, and the nodes to be reordered in blue. We model this as a cycle too, instead of inserting a dummy node, we insert \( a_1 \) and \( a_2 \) with an unbreakable connection between them, and use those to open the resulting cycle to obtain a path. The TSP solver needs to be modified slightly to avoid breaking the link between \( a_1 \) and \( a_2 \). Figure 2.5 shows that the red links are broken, and replaced by the blue links. In the 2-OPT algorithm, we simply disallow breaking the link between \( a_1 \) and \( a_2 \), so we treat the exchange as if it did not improve the solution.

### 2.3. Icicle Tree

The adjacency matrix visualization, as described in section 2.1, does not visualize the hierarchy in the network. This is an issue, because the hierarchy is an integral part of our data.

We chose to visualize the hierarchy explicitly, that is, simply show the tree. Figure 2.7 shows an adjacency matrix at two different levels, with the hierarchy to the sides. On the right, the hierarchy is visualized as a standard node-link diagram. However, our application uses a different method: an icicle tree [KL83], shown on the left of the matrices in figure 2.7. The main advantage of an icicle tree is that it works well with our usage of summary nodes in matrices. The figure also illustrates that parent nodes in the icicle tree have the same size as summary nodes in the matrix. This property makes it useful as an augmentation to the matrix visualizations in section 2.1.

Lastly, an icicle tree presents a convenient way to implement interaction. It allows selecting nodes on every level of the hierarchy, and is also intuitive to the user.

### 2.4. Edge Bundling

As mentioned before, an adjacency matrix is an effective graph visualization method, but is not able to intuitively represent the connections from one node to multiple nodes. Furthermore, paths between nodes are very hard to distinguish. Because of this, we implemented a visualization that offers a different perspective on the same data, while also visualizing the hierarchy in the network.

We chose to implement a variation on a node-link diagram. However, we only render edges, and use the matrix to provide nodes. Figure 2.8 shows an adjacency matrix, with our edge rendering on the side. The hierarchy is shown in gray, and is used to provide control points for the Beziér [FHK02] curves used to draw the edges. The result of using Beziér curves is that edges with a common parent tend to be close to each other, effectively bundling them.
A Beziér curve [FHK02] is based on a number of control points, \( p_1, p_2, \ldots, p_n \). The curve goes from \( p_1 \) to \( p_n \), but does not necessarily pass through the other control points. Figure 2.9 shows what a Beziér curve looks like, for \( n = 5 \).

This method we implemented is called hierarchical edge bundling, and was described by Holten [Hol06], but our version is slightly modified. As previously mentioned, we use the node locations provided by the matrix. The advantage of this method is that it neatly links the matrix visualization to our edge rendering visualization. The matrix is sorted, so nodes with a similar neighborhood are already close to another. This means that edges inside clusters generally result in many short lines, which is exactly the desired behaviour. A small illustration can be seen in figure 2.8. This figure also shows the hierarchy, which is used to provide control points for the Beziér curves. Like the solution by Holten, we do not use the local root node as a control point, unless discarding it would cause the number of control points to go below 3. The result is a variable number of control points \( n \), dependent on the path an edge takes through the tree. The resulting Beziér curve is of degree \( n - 1 \). This path is obtained as shown in algorithm 2.1, by following parent pointers until a common ancestor is found.

\begin{algorithm}
\begin{algorithmic}
  \Function{getPath}{\( n_1 \), \( n_2 \)}
  \Comment{Assume \( n_1 \neq n_2 \) and depth\( (n_1) = \text{depth}(n_2) \)}
  \State \( P_1 = \emptyset \)
  \State \( P_2 = \emptyset \)
  \State \( a = n_1 \)
  \State \( b = n_2 \)
  \While{\( a \neq b \)}
    \State \( P_1 = P_1 \oplus \{a\} \) \Comment{\( \oplus \) is concatenate}
    \State \( P_2 = \{b\} \oplus P_2 \)
    \State \( a = \text{parent}(a) \)
    \State \( b = \text{parent}(b) \)
    \If{\( |P_1| + |P_2| = 2 \)}
      \State \( P_1 = P_1 \oplus \{a\} \)
    \EndIf
  \EndWhile
  \State \Return \( P_1 \oplus P_2 \)
\end{algorithmic}
\end{algorithm}

Holten notes that his method does not work well when too many points are colinear, which is exactly our situation. To avoid this issue, all control points are put on a curve. The farther an original control point was from the center of the visualization, the higher it is placed. The result is that the control points are no longer colinear. Figure 2.10 shows that the results are significantly less cluttered with control points on a curve. The function we used to compute the adjusted height of the control points is shown below, in equation 2.11. It modifies the \( y \) attribute of a point \( p \). In that formula \( W \) and \( H \) correspond to the width and height of the region to draw in, and \( s \) is a scaling factor.

\[
p_y = p_y + sH \left| \frac{1}{2} \frac{p_x}{W} \right|^2
\]  

(2.11)
Figure 2.10  Top: control points on a blue straight line, bottom: control points on a blue curve. The area inside the red rectangle is less cluttered when using a curve. The hierarchy is shown in gray, and the root node is not shown.

Figure 2.11  Edge bundling with two graphs. Edges are rendered for each graph separately, and the rasterized result is merged back together and colorized to indicate which graph had the higher value.

The edge bundling method bundles edges, which causes many edges to be drawn on top of each other in bundles. Without special care, this causes the loss of all detail in the bundles. We avoided this by using additive blending, along with a normalization step. Using additive blending adds the color value of the new line to the value that was already in the framebuffer. In many situations this will cause the color value to go above $(1,1,1)$ (lines are drawn in grayscale), which corresponds to white. This means a normalization step is required, to make sure only the absolute largest value hits pure white. The normalization function used is shown below in equation 2.12, which replaces the original $r,g,b$ triplet with $r',g',b'$. The normalization function uses a logarithmic scale, otherwise only the most overdrawn areas would be visible. In the equation below, the scaling function defined in equation 2.1 is used. Note that $M$ indicates the largest value of all pixels.

$$r' = g' = b' = \langle r \rangle_M$$  \hspace{1cm} (2.12)

Finally, we add the ability to show differences between graphs. We do this by rendering the two

Figure 2.12  Edge bundling as seen in our application using a randomly generated dataset.
graphs separately. This results in two images, one for each graph. Overlaying these two images results in two values per pixel. This is a problem we saw before, when trying to compare two matrices. To decide the final color, we chose one of the matrix visualization options described in section 2.1 that do not require any additional geometry. Because the saturation method, see equation 2.8, is able to encode all information in one pixel, it is our visualization of choice. In equation 2.8 the value $v_1$ is the value corresponding to the first graph, $v_2$ is the value corresponding to the second graph, and $H = \max(M_1, M_2)$ is the largest value in both graphs. This whole process is shown in figure 2.11, and the final result can be seen in figure 2.12.

## 2.5. Clustering

The main point of edge bundling is that a hierarchy is used to bundle edges. However, some of our datasets have no hierarchy, or only a very shallow one. This means the edge bundling visualization has very little information to work with. A second issue is that, without hierarchy, our matrix visualization cannot provide an overview of the data. To fix this, a clustering method was implemented to add or extend a hierarchy.

Clustering, in general, behaves as shown in figure 2.13. Clustering works in passes, one pass clusters the children of each node on a level separately. That is, when one node on a given level has too many children, all nodes on that level are clustered into the appropriate amount of clusters. The nodes that do not need splitting are simply clustered into one cluster. For the nodes with too many children, the process works as follows. The node with too many children, from now on called “root”, will be the root of our subtree. We will add a hierarchy level between the root and its children. In that new level, $k$ new nodes are added, where $k$ is the number of clusters, determined by the clustering algorithms described below. The process of finding levels that need clustering repeats until no nodes have too many children, which results in a deepened hierarchy. This process is explained in more detail by algorithm 2.2.

Clustering happens on multiple networks at once. One of the requirements for our program is that the hierarchies between networks match, which was also mentioned in section 1.1. The used clustering algorithms are all slightly randomized, so separate clustering would not yield compatible hierarchies. The clustering method as used on line 7 in algorithm 2.2 uses a random starting position for better results. The problem with this is that running the clustering method on the same dataset twice will not always yield the same hierarchy. To avoid this issue, two networks can be clustered together, at the same time. So where the neighborhood of node $i$ is defined by column $i$ in the matrix, it will now be defined by a concatenation of the $i$th columns in all matrices. So when comparing $n$ networks, this will result in $n$ columns at position $i$. The final result is a vector.
Algorithm 2.2 An algorithm that clusters a tree, see figure 2.13. Input is a list of levels \( L \), and a node size limit \( t \). The level-list contains, at each level, all nodes at that depth.

```plaintext
function clusterTree(L, t)
    for \( i \in [0 \ldots |L|] \) do
        // For every level, starting from the leaves
        \( l = L[i] \) // \( l \) is the list of nodes at level \( i \)
        if \( \exists n \in l \) (\( |\text{children}(n)| > t \)) then
            \( L = \emptyset \)
            for \( n \in l \) do
                \( C = \) clustering of children\((n)\) into \( \lceil|\text{children}(n)|/t\rceil \) clusters
                \( M = \emptyset \) // A list of new nodes, one for each cluster in \( C \)
                for \( c \in C \) do
                    \( m = \) new node // A blue node as shown in figure 2.13
                    \( \text{children}(m) = c \)
                    \( M = M \cup \{m\} \)
                    \( \text{children}(n) = M \)
                    \( L = L \cup M \)
                insert \( L \) into \( L \) at position \( i \) // The next \( l \) will be equal to the current \( l \)
```

of size \( n \cdot m \) where \( n \) is the number of networks that are used, and \( m \) is the width and height of the matrix.

We chose the concatenation method over computing the average, because computing the average can group the wrong elements. For example, we have node \( v \), node \( w \), and node \( k \). The connection strength from node \( v \) to node \( k \) is 0 for network one, and 1 for network two. The connection from node \( w \) to node \( k \) is 0.5 for both networks. If we were using the average, the neighborhoods of \( v \) and \( w \) with respect to \( k \) are identical. The concatenation method, however, results in two dimensions that differ by 0.5.

To do the clustering, three clustering methods were implemented. KMeans, TSP, and Left-Right split. Some datasets have disjoint meshes for the two hemispheres in the brain, which can be detected. This is what the Left-Right split does, it always splits the brain in two. In most cases this is a good starting point, so this is done by default for data that does not have a left-right split at the top level of the hierarchy. Afterwards, we use the other methods to cluster the regions in the newly identified hemispheres into more pieces. For that we implemented two options, KMeans and TSP. Initially we started with KMeans, because of its speed and ease of use. Later we implemented the slower TSP method, which provides higher quality results. The methods themselves are described in more detail below.

KMeans KMeans [M+67] is a fast clustering algorithm, which is also very easy to implement. KMeans works in Euclidian space, and depends on the triangle equality. Because this method works in Euclidian space, each node needs a position. This position, however, does not need to be in only three dimensions, which allows clustering on neighborhoods as well. One method, is simply using the 3D location of each node, and cluster that way. Our application uses the neighborhood of a node instead. The neighborhood of a node is simply a column in the adjacency matrix, which will be the node’s location in \( m \)-D space, where \( m \) is the number of nodes.

Because we need to cluster several networks at the same time, the node’s location needs to be modified slightly. Instead of using a vector of size \( m \), we use a vector of size \( n \cdot m \) instead, simply a concatenation of the columns in the \( n \) matrices we are clustering. This concatenation method is described above in more detail.

Note that we can not use the distance metric as defined in equation 2.9, because that equation does not necessarily adhere to the triangle inequality, which KMeans needs. Instead, we use the Euclidian distance.
The $k$ in KMeans stands for the number of clusters to create. In our application that is simply an input, we chose not to implement any method that automatically detects the number of clusters.

**TSP**  Our traveling salesman-based clustering uses a modification of the ordering method described in section 2.2, as described by Climer and Zhang [CZ06]. Instead of adding exactly one dummy node, the idea is to add several dummy nodes. Assuming we are using a good distance metric, the distance between nodes in two different clusters is relatively high. More generally, the borders between clusters will be the most “expensive” edges in the cycle. This means that, if we add dummy nodes whose distance to every other non dummy node is 0, these dummy nodes will be placed at cluster boundaries. Clustering into $k$ clusters means we have to use $k$ dummy nodes, because $k$ cuts will partition a cycle into $k$ pieces.

For this clustering method, which does not rely on the triangle inequality, the Tanimoto [Tan57] distance, as defined in equation 2.9, can be used. TSP clustering generally generates higher quality results than the KMeans clustering, because the Tanimoto distance focuses more on neighborhood than the Euclidian distance.

Like the KMeans solution, this solution uses $k$ as an input variable, and does not attempt to detect how many clusters would be needed.

**Left-Right hemisphere split**  The datasets contain a mesh which consists of two parts, one for each hemisphere, and these parts are not connected with any triangles. In other words, the brain mesh consists of vertices and triangles. Every triangle connects three vertices, but there are no triangles that connect vertices which belong to different hemispheres. This allows the use of a flood-fill algorithm to detect which vertices belong to which hemisphere. This works by choosing a vertex $v$, and see which other vertices are reachable via the faces that connect to vertex $v$. This can simply be repeated, until no new vertices are found. Then, all vertices that were unreachable will belong to the “other” hemisphere. This technique always divides the dataset in two.

## 2.6. Pre-selection

As mentioned in section 1.4, it was expressed by neuroscientists that it would be interesting to select a few regions, and ignore the rest of the network. To fulfill that request, a selection method was implemented that preserves the hierarchy where possible. The effect of the selection method is shown in figure 2.14.

A formal definition of the pre-selection algorithm is shown in algorithm 2.3. The input to this algorithm is a set of nodes per level, and an integer $k$, which is the level at which the selected nodes are found. Node selection works as follows: if a node is selected, do nothing. Otherwise, if one of its ancestors is selected, also do nothing. If none of the ancestors of a node are selected, find the nearest ancestor at a level higher than the level containing the selected nodes. This ancestor will have a merge node, or one will be created. When the algorithm is done, the merge node will represent all other nodes that shared its parent.
Algorithm 2.3 Pre-select nodes in a tree as shown in figure 2.14. Every node that is not selected will be assigned to a “merge” node at the same level as that node. The level on which the selected nodes are found is $k$. The other input is $L$, which indicates the list of levels.

```plaintext
function mergeNodes(k, L)
    for $l \in [k...0]$ do // For levels $k$ and below. Level 0 corresponds to the leaves
        $N = L[l]$ // $N$ is the list of nodes on level $l$
        for $n \in N$ do // For all nodes on this level
            if $\neg$selected($n$) then
                $p = n$
                while $\neg$selected($p$) $\wedge$ level($p$) $< |N|$ $\wedge$ $\neg$hasMergeNode($p$) do
                    $p = parent(p)$ // $p =$ nearest ancestor that remains unchanged (gray in figure 2.14)
                if $\neg$selected($p$) then
                    while level($p$) $>$ $l$ do // Go down the tree again, only visiting merge nodes
                        if children($p$) contains a merge node $m$ then
                            $p = m$
                            else // This branch will only be taken when $n = p$
                                $m =$ new merge node
                                children($p$) = children($p$) $\cup$ {$m$}
                                $p = m$
                            // $p$ now contains the merge node at level $l$ (blue boxes in figure 2.14)
                    add $n$ to $p$
            for $l \in [k...0]$ do
                $N = L[l]$
                for $n \in N$ do
                    if hasMergeNode($n$) then // Remove non-merge nodes with merge node sibling
                        $M =$ children($n$) except mergeNodes
                        children($n$) = children($n$) $\setminus$ $M$
                        $L[l - 1] =$ $L[l - 1]$ $\setminus$ $M$
        end for
    end for
end function
```
The two images show the brain regions on different levels of the hierarchy. The image on the left shows the bottom level of the hierarchy, with 500 nodes. The image on the right shows the hierarchy level just above the lowest one, which has only 100 nodes.

### 2.7. 3D Anatomical View

The main disadvantage of the edge bundling visualization in section 2.4 and the adjacency matrix visualization in section 2.1, is that they both discard all anatomical information. During early user testing, this was identified as an issue, so we implemented a visualization focused on providing this information. This visualization renders a 3D mesh that represents the cortex, which was distributed with the data. Each vertex in the mesh has an associated node for each level of the graph. When highlighting a node in the graph, the vertices that correspond to that node will be highlighted as well.

A rendering of the mesh can be seen in figure 2.15. The mesh potentially consists of two disconnected parts, one for each hemisphere. These two parts were identified earlier using the left-right split clustering, see section 2.5. The two groups of vertices can then be moved away from each other slightly. This allows the user to see what is going on between the two hemispheres.

Each region needs to be visually identifiable, so neighboring regions must have a different color. This problem comes down to the graph coloring problem [JT11], which is NP-complete. We chose a different solution, which is to attempt to generate a unique color for each region. This is guaranteed to be suboptimal, and prone to assigning two similar colors to neighboring regions, but is far easier to implement. Even though it is easier, it is still not trivial, since our datasets have up to 4000 nodes at the lowest level. To generate that many different colors, we need many distinct and well spaced numbers in the range $[0, 1)$. To generate those numbers, we use the following formula:

$$ f(i) = (i \cdot \alpha) \mod 1 $$

where the value $i$ is an integer which ranges from 0 to $n$, where $n$ is the number of colors that need to be generated. If we choose $\alpha = \frac{p}{q}$, with $p, q \in \mathbb{N}$, the value for $f(i)$ repeats every $q$ times, so $q$ must be at least $n$. The problem with using $\alpha = \frac{1}{n}$ is that consecutive values for $i$ have very similar colors.

To fix the issue of nearly overlapping colors, we can use a different value for $\alpha$. The golden ratio, $\varphi = \frac{\sqrt{5}+1}{2} \approx 1.618034$, is the real number which is worst approximated by rational numbers [And97]. The result is that, if we take $\alpha = \varphi$ in equation 2.13, $f(i)$ never repeats. This is, however, not yet unique to the golden ratio, this is true for any irrational number. However, the golden ratio is the most irrational number, which means that, for any $i$, $f(i)$ from equation 2.13 using the golden ratio is farther away from an approximating fraction than $f(i)$ using any other number. The result is that the golden ratio yields a more evenly spaced distribution of values. The fact that the golden
Figure 2.16  The result of equation 2.16. The bottom row indicates the resulting color. The far left side corresponds to \( f(i) = 0 \), the far right corresponds to \( f(i) = 1 \). Note that the value for hue changes from 0 to 1 four times.

The golden ratio is the most irrational number can be expressed more formally as follows. For all \( \alpha \in \mathbb{R}, \alpha \neq \varphi \) and for all reduced fractions \( \frac{p_1}{q_1} \) and \( \frac{p_2}{q_2} \) holds that [LNRR02]:

\[
\left| \frac{p_1}{q_1} - \varphi \right| \leq \left| \frac{p_2}{q_2} - \alpha \right| \Rightarrow q_1 > q_2
\]  

(2.14)

In other words, to approximate the golden ratio as well as another real number \( \alpha \), the fraction used to approximate the golden ratio needs to be more complex than the one needed to approximate \( \alpha \). In this context, complexity corresponds to the size of the denominator of the reduced fraction.

The golden ratio being the most irrational number makes it an appropriate choice for a method called Fibonacci hashing. The name is based on the way the golden ratio \( \varphi \) is approximated using the Fibonacci sequence [Tat05]:

\[
\lim_{i \to \infty} \frac{F(i+1)}{F(i)} = \varphi
\]  

(2.15)

Fibonacci hashing comes down to formula equation 2.13 with \( \alpha = \varphi \), as described by Preiss [Pre98]. Preiss [Pre98] remarks that every new value sampled from \( h(i) \) divides the interval into which it falls according to the golden ratio. This property results in very even spacing of values.

Fibonacci hashing uses the property that the numbers \( f(i) \), for \( i = [0, 1, 2, 3, \ldots] \), are evenly spaced in the range \( [0, 1) \), which is exactly what we need too. To convert these evenly spaced numbers to colors, all that is needed is the HSV color model. A good result can be obtained by mapping \( f(i) \) to the hue variable, but this leaves saturation and value unused. To increase color diversity, the following function is used to generate colors:

\[
h = 4f(i) \mod 1 \\
 s = (\lfloor 4f(i) \rfloor \mod 2) \cdot 0.4 + 0.4 \\
v = (\lfloor 2f(i) \rfloor \mod 2) \cdot 0.4 + 0.4
\]  

(2.16)

The result of equation 2.16 can be seen in figure 2.16. Note that \( s \) and \( v \) have only two possible values, 0.4 and 0.8. The values themselves are not too important, but it is important that they are fairly far apart, and that neither ever reaches 0. If the saturation \( s \) is 0, the hue no longer matters, which means the result is almost useless. The same is true for the value \( v \), if that hits 0, the result is always black, no matter the value for saturation or hue.

### 2.8. 3D Edges

The edge bundling visualization and the matrix visualization can be too abstract, and they disregard the anatomical locations of the nodes as well. To remedy this, the 3D surface rendering was added to provide context. We believe that a visualization that renders connections without ignoring anatomical locations can be a good addition to our application. This visualization works by drawing lines in 3D, each line corresponding to a connection. There are several ways to do this, one is simply drawing lines “on top” of the brain, like the solution by Xia et al. [XWH13] in figure 1.6. Another is working with transparency, by drawing the edges straight through the brain mesh, as
Figure 2.17 The springs that control the chain. In the left part, red arrows indicate springs that push the control points away from the surface. On the right, red arrows indicate structural springs that only pull. Blue arrows indicate springs that try to prevent bending.

Figure 2.18 The iteration process for our line renderer. It starts with an initial solution which follows the surface, and iterates towards a nicer solution.

shown in figure 1.4. However, we chose a third solution, which, like Foucher et al. [FVC+05] in figure 1.5, draws the lines around the brain instead. This method has some advantages over the two other solutions, which are basically ways to combine 2D edges with a 3D mesh. The advantage of the method by Foucher et al. is that it is easier to see which regions are connected because the connections themselves are also drawn in 3D.

Our method, with lines around the brain, does present some difficulties during implementation. In our application we render the brain mesh at full opacity, so if an edge were to clip through the surface, it would become hidden. This would make it difficult for the user to follow the edge. Edges clipping through the surface is a real issue, due to the extremely curved nature of the brain surface. To solve this issue, we implemented a force directed layout algorithm [FR91]. The main advantage of this method is that the problem can be intuitively described. The lines should not clip through the surface, so just make the surface push the lines away. Similarly, the lines should be as short as possible, and should curve smoothly, which can be expressed by making the lines behave like rubber bands.

Our force directed layout algorithm uses a spring chain. Every edge is represented by a number of control points, and is drawn by simply connecting them with a straight line. The surface exerts forces on these control points, and control points themselves want to be close to each other. The structure of these edges is shown in figure 2.17, which shows three types of springs:

- The first spring type is the structural spring, shown in red on the right side of figure 2.17. This spring simply pulls, trying to bring neighboring control points closer together.

- The second type of spring, shown in blue in figure 2.17, pulls too. They are connected to control point \( n_i \), and a dummy control point at \( (n_{i-1} + n_{i+1})/2 \). These springs make sure the spring chain does not bend too sharply, which would make the 3D edge hard to follow.

- The third spring type, in dark red, pushes the spring chain away from the surface. These springs are very stiff, and have a target length \( t \). If a node is farther away than \( t \), these springs remain inactive, as indicated by the sometimes absent dark red arrows in figure 2.17.

Our force directed layout algorithm is an optimizer. That is, it requires an initial state, and iterates towards a local optimum. This process is illustrated in figure 2.18. The initial state is obtained by using a shortest path algorithm which finds a path from one vertex on the surface to another vertex.
on the surface. This path consists of vertices on the surface, so it seems like an adequate initial state. However, since we use a force-based technique, there may be stability issues. The surface springs may push too hard, which causes the spring chain to be pushed too far away to converge back to a stable state in a reasonable timeframe. To avoid this issue, we simply move the initial control points outwards in the direction of the normal at that point on the surface, at the risk of intersecting the surface.

**Equation 2.17** describes the formula we used to compute the spring forces. It uses two input points, \( p_1 \) and \( p_2 \), and parameters \( t \) for the rest length and \( c \) for the spring strength. The force resulting from \( \Delta(p_1, p_2, t) \) makes \( p_1 \) go toward \( p_2 \), with the strength of that force depending on the distance between the two. The constant \( t \) describes the rest length, which is the length the spring wants to have. If \( ||\delta(p_1, p_2)|| < t \), the spring will push \( p_1 \) away from \( p_2 \) instead of pulling \( p_1 \) towards \( p_2 \). The springs between control points can be modeled by putting a spring like equation 2.17 between successive control points. When computing surface forces, however, not all springs apply at the same time, and the set of springs that do apply changes between iterations. To compute the forces between the surface and the control points, **algorithm 2.4** is used, which finds the springs that need to be used during the current iteration. An octree was used to find the vertices that are sufficiently close to a control point to warrant the inclusion of a surface spring.

\[
\delta(p_1, p_2) = \text{pos}(p_2) - \text{pos}(p_1) \\
\Delta(p_1, p_2, t) = \frac{\delta(p_1, p_2)}{||\delta(p_1, p_2)||} (||\delta(p_1, p_2)|| - t) \cdot c \\
\text{accum}'(p_1) = \text{accum}(p_1) + \Delta(p_1, p_2, t)
\]  \hspace{1cm} (2.17)

As shown in the above formula, forces are added to an accumulator first, instead of being applied directly. This is because each control point is affected by multiple springs, and the forces for all springs should be computed from the same state.

**Algorithm 2.4** Computing surface forces. All vertices that are close enough are considered, and are found using an octree. The arguments are a list of control points, a mesh, and a target distance \( t \).

```plaintext
function computeSurfaceForces(line, mesh, t)
    for p \in line do
        A = \{v \in mesh | distance(p, v) < t\}
        f = (0, 0, 0)
        for v \in A do
            f = f + \Delta(p, v, t)  // See equation 2.17.
        accum(p) = accum(p) + f / |A|

```

After computing all springs, the following is executed for all control points \( p_i \). In this formula \( f \) indicates the friction, and \( h \) indicates the timestep.

\[
\text{speed}'(p_i) = \text{speed}(p_i) \cdot (1 - f) + \text{accum}(p_i) \\
\text{pos}'(p_i) = \text{pos}(p_i) + \text{speed}(p_i) \cdot h \\
\text{accum}'(p_i) = (0, 0, 0)
\]  \hspace{1cm} (2.18)

Until now, only one iteration was described. However, to compute nice looking lines, multiple iterations are required. There are several possible approaches, such as iterating until a stable state is found. Instead, we chose to iterate a fixed number of times. The advantage of using a fixed number is that even if the computation “explodes”, the iteration process stops quickly. The main disadvantage is that lines that converge quickly take more time than necessary. This issue can be fixed by iterating to a stable state, with a limit on the number of iterations. However, detecting a stable state can be difficult in case of vibration, or lines that converge extremely slowly. Very slowly converging lines can simply be stopped, because more iterations do not significantly influence the end position. We chose to stop after a certain number of iterations, since reaching a local optimum can take a long time, and is not necessary for this application.
To reduce the number of required iterations, the computations use momentum. That is, control points in the spring chain have a speed attribute. The result is that fewer iterations are required to obtain the same result. Without momentum, only the first few iterations cause significant movement. With momentum, large forces computed during one iteration slowly dissipate during subsequent iterations, instead of disappearing instantly the next iteration. This means the use of momentum makes the whole process faster. However, this does further exacerbate the stability issues our method already has. Most force directed layout methods have stability issues, and they are only worsened by using momentum. The problem is that during some iteration a control point in the chain can get very close to the surface of the brain, which causes it to be pushed back very hard. It will then take many iterations to get back to a stable situation, or maybe it will never reach a stable state again. We slightly alleviated the issue by introducing friction, but avoiding this issue mainly comes down to properly tuning the strength of each spring, the friction, and the constant which defines the desired distance to the surface.

Instability is a common issue with force based methods, especially when using momentum. This issue can be alleviated by using the midpoint method [CC98] as our integration method, which is more stable than the simpler Euler method [Atk08], as used in equation 2.18. The formal definition of the Euler method is also shown in equation 2.19. Other methods, like the fourth order or higher Runge-Kutta methods [But87], are more stable, but slower too. It is our opinion that the midpoint method is a good compromise between speed and stability, so it is used in our application. The midpoint method is defined by equation 2.20, and uses a function \( f(t, y) \) where \( t \) is the time, \( h \) is the timestep, and \( y \) is the state. The formula describes how the state \( y_{n+1} \) is computed from the previous state \( y_n \), which was at time \( t_n \).

\[
y_{n+1} = y_n + h \cdot f(t_n, y_n) \tag{2.19}
\]

\[
y_{n+1} = y_n + h \cdot f \left( t_n + \frac{h}{2}, y_n + \frac{h}{2} f(t_n, y_n) \right) \tag{2.20}
\]

In our case, \( f(t, y) \) is defined in equation 2.18. For the half step, the \( \frac{h}{2} f(t_n, y_n) \) part of equation 2.20, we compute the next step as in the Euler method from equation 2.19, but instead of using the normal timestep \( h \), we use \( \frac{h}{2} \). For the next step, we compute another accumulator using the state defined by the half step, but apply the formula equation 2.18 on the original state, that is, the state before doing the half step.

The time complexity of our iterative solution is defined by two main sources: computing the forces among control points, and computing the forces between control points and the surface. Computing forces between control points requires only linear time, scaling with the number of control points that define the line. The time complexity for computing the surface forces, as described in algorithm 2.4, is more complicated. The time complexity of that component is \( O(nX) \), where \( n \) is the number of control points, and \( X \) is the time it takes to query all vertices in radius \( l \) around a point \( p \). To limit the size of \( X \), an octree was used to do the querying. However, it is hard to give a running time for this octree query, because in the worst case, all points will need to be returned, so the size of \( X \) depends strongly on the distribution of vertices in the mesh.

The final result is shown in figure 2.19. For better shape perception of the lines we added lighting. To use lighting, a normal is needed for every point on the line. However, a line does not have a unique normal. To choose a suitable normal on a line from \( p_1 \) to \( p_2 \) the following formula can be used, where \( L \) is the light position.

\[
\begin{align*}
R &= p_2 - p_1 \\
U &= L - p_1 \\
N &= \frac{(R \times U) \times R}{\| (R \times U) \times R \|} \tag{2.21}
\end{align*}
\]

In this equation, also shown in figure 2.21, \( U \) is the vector from a point on the line, such as \( p_1 \), to the light position, and \( R \) is a vector in the direction of the line. To compute the desired normal, the cross product can be used as shown. The first cross product, \( R \times U \) results in a vector perpendicular
Figure 2.19  Edge rendering in 3D, without lit edges.

Figure 2.20  Edge rendering in 3D, with lit edges.

Figure 2.21  Computation of a normal for a line from $p_1$ to $p_2$, see equation 2.21. $L$ indicates the light position. The dotted, red, line indicates that it points outside the paper. That is, it is perpendicular to the lines $U$ and $R$.

Figure 2.22  A line is lit like a thin cylinder. The normal at the intersection of a ray with the surface of the cylinder is always in the plane defined by $R$ and $L$. 
to both $R$ and $U$. However, this vector is no longer in the plane defined by the vectors $U$ and $R$. Using the cross product with $R$ again results in a vector that will be in the desired plane. The result, with lighting, can be seen in figure 2.20. The illumination is based on the assumption that the line is a thin cylinder, as shown in figure 2.22. For any light ray that is cast from $L$ to any point on the line segment between $p_1$ and $p_2$ the normal will be the same, and perpendicular to the surface of the cylinder.

## 2.9. Artificial Datasets

We provide two different artificial datasets. One is a purely random dataset, while the other only replaces the connectivity information of an input dataset. The purely random dataset is mostly used for testing. The other is useful for an end user, because it provides a network to compare with the original network.

### 2.9.1. Distance Based

We noticed that many brain regions are connected to their neighbors. We believe that these connections are likely less interesting. Because of that, we provide a distance based connectivity matrix to compare with. This connectivity matrix simply connects nodes that are close to each other. It can be generated based on another dataset, so it always has a matching hierarchy.

Every node has a 3D location, defined by its vertices. The adjacency matrix we generate is defined by algorithm 2.5. This algorithm takes three variables as input, the list of nodes, a corresponding adjacency matrix to replace, and a constant $c$. The algorithm assigns a value to each cell based on distance. The constant $c$ indicates how far a node $n_j$ can be away from the current node $n_i$ while still being connected.

**Algorithm 2.5** An algorithm which generates a distance based adjacency matrix.

```plaintext
function modifyAdjacencyMatrix(nodes, matrix, c)
    for i = 0 to |nodes| exclusive do
        n = nearestOtherNode(nodes[i], nodes)
        d = distance(nodes[i], n)
        for j = 0 to i exclusive do
            d2 = distance(nodes[i], nodes[j])
            v = max \(0, 1 - \frac{d2}{c \cdot d}\)
            matrix[i][j] = v \cdot matrix[i][i]
            matrix[j][i] = matrix[i][j] // The matrix is symmetrical
```

The main advantage of this method is that when comparing the generated matrix with the original, the differences will be interesting. For example, what if the distance based matrix does not connect two regions, but the actual matrix does. This would indicate a connection between two distant regions, which can be interesting. Similarly, if the distance based matrix does connect two regions, while the actual matrix does not, that could be interesting as well because it would mean that two regions which are very close are not actually connected.

On line 8 in algorithm 2.5, the generated value is multiplied with the value that was found on the diagonal. The reason for this is that the generated values will automatically have the same maximum value as the original matrix, which makes comparisons easier. This is, of course, assuming the value at the diagonal is not zero, and useful. If the value at the diagonal were zero that would mean that node would not be connected to itself, but in all tested datasets the values on the diagonal are non-zero.
2.9.2. Random

To more easily test our methods, datasets with a simple structure can be useful. To generate an adjacency matrix with a simple structure, we used a method which puts boxes at the diagonal. The size of those boxes is random, as well as the degree to which they overlap. Figure 2.23 shows what this technique looks like. To make things more interesting, the boxes are not entirely filled. For every cell that overlaps a box, a random value is generated. This random value is biased so that cells closer to the diagonal are more likely to get a high value. This whole process is described in more detail in algorithm 2.6.

**Algorithm 2.6** Generating a random matrix of size $n \times n$, with the structure shown in figure 2.23. The effects of the variables “size” and “overlap” are shown in that figure as well, in blue and red respectively.

```plaintext
function GENERATERANDOMMATRIX(n)
    M = empty $n \times n$ matrix
    for $i = 0$ to $n$
        $M[i][i] = 100$
    size = 0
    $i = 0$
    while $i < n$
        if $i > 0$
            overlap = $(\text{size}/2) \cdot \text{random()}$ // $0 \leq \text{random()} < 1$
            $i = i - \text{overlap}$
        size = min($n - i, \lfloor \text{random()} \cdot n/5 \rfloor$)
        for $j = i$ to $i + \text{size}$
            for $k = i$ to $i + \text{size}$
                $f = \frac{(j - k)^2}{\text{size}^2}$
                $M[j][k] = \max(0, 100 \cdot (1 - f) + f \cdot \text{random()} \cdot 100)$
                $M[k][j] = M[j][k]$
        return $M$
```

**Figure 2.23** Random matrix generation. The diagonal is filled with boxes of random size (blue). The amount of overlap is random, and shown in red. Every box overlaps at most 2 other boxes.
Figure 3.1 The way data arrives at our application. The data sources are shown on the left of this flow chart. Blocks with a gray background are parts of our implementation, and are described in chapter 3. Blocks with a blue border are file formats, and are described in appendices A to D.

## 3. Implementation

This chapter describes implementation details to supplement the information found in chapter 2. The information mentioned here provides insight into our implementation as well as some of the hurdles that were overcome. This chapter also describes how to interact with our tool, in section 3.3.

The implementation is split into three parts, a data conversion script for the datasets from Barcelona, a larger and more general data conversion and preprocessing program, and finally a plugin in vIST/e [Ein15]. The general structure is as shown in figure 3.1. Note that our vIST/e plugin requires its input files to be in the non-standard XHGFZ format, see appendix D. This means that the data preprocessor is a required component when using our tool.

**Data converter** This script converts data in the gpickle format (appendix B), to the cocomac format (appendix A).

**Data preprocessor** This program converts data from the cocomac format, described in appendix A, and data from Maastricht, described in appendix C, to our own XHGFZ format, described in appendix D.

Other than converting data, the preprocessor supports the following tasks:

- Clustering using KMeans or TSP on a group of XHGFZ files. See section 2.5.
- Output a text representation of the hierarchy inside an XHGFZ file.
- Validating an XHGFZ file.
- Filtering geometry, by throwing away faces with a surface area of zero.
- Removing the hierarchy from an XHGFZ file.
- Generating a pseudo random dataset, see section 2.9.
- Replacing the connectivity matrix of an XHGFZ file with a distance based matrix, see section 2.9.
- Pre-selection, see section 2.6

**vIST/e plugin** vIST/e [Ein15] is a framework for medical visualizations. We believe that users will appreciate being able to work with a familiar tool.

Our vIST/e plugin contains the following modules, which are all rendered using OpenGL:

- Matrix sorting
- Matrix rendering
- Icicle graph
- 2D Edge rendering
- 3D Surface rendering
- 3D Edge rendering
In the sections below, we will discuss our methods for rendering matrices, and how we optimized them. We will also discuss some technical details of our edge bundling visualization.

3.1. Matrix Rendering

Matrix rendering has two cases, leaf rendering and non-leaf rendering. These two are not the same, because at non-leaf levels, nodes are not necessarily square. This means the technique used for leaf rendering cannot be used there.

**Leaf rendering** The matrix is saved to a non-normalized dual channel texture, of type GL_TEXTURE_RECTANGLE. Non normalized means the values per pixel are not in the range [0,1], but can be any pair of floating point values. The textures used are limited to 256 × 256, and are tiled to cover the entire matrix. The advantage of this method is that there is no need for a very large contiguous section of memory. The values saved in the texture represent the actual connection strengths of an edge, so the methods in section 2.1 can be used directly by the fragment shader. Some of the described matrix visualizations use geometry, which is generated implicitly by the fragment shader. This works by checking what part of the per-cell geometry the current pixel belongs to, and assigning a color based on that information.

The shader that computes the Barchart option can be seen in algorithm 3.1. Other options described in section 2.1 use a different getColorInCell function. The shader in algorithm 3.1 is a fragment shader, so it runs once for each pixel covered by the geometry. In our case, we render a large quad, so only the pixels inside that quad will be passed through the fragment shader. The first part of the main method, from line 23 to line 28, consists of figuring out which cell in the matrix is below this pixel. Then, the values in that cell are obtained by the texture2DRect call. The variable pos contains the position inside the current cell. In the bottom left corner pos = (0,0), in the bottom right corner pos = (1,0), etc. This variable is used to generate geometry inside matrix cells. Afterwards, the actual color is computed, and slightly modified with a gradient right after (line 35). This gradient allows the user to see the boundaries between cells without them being explicitly drawn. The gradient was chosen to maximize the difference in color between neighboring cells, while keeping the original color as much as possible. Finally, the color is converted to rgb, and assigned to the output variable gl_FragColor.

The input to the shader in algorithm 3.1 consists of two types. The first is an in variable, texco, which is sent from the vertex shader. These values are interpolated, a quad is rendered with texture coordinates from 0 to 1. The vertex shader passes these on to the fragment shader, which gets interpolated values dependent on the position of the fragment. So a fragment in the center will have texture coordinates (0.5,0.5). The shader also gets uniform variables. These variables are global, so they are the same for all fragments. One of those variables is threshold, which can be changed at runtime at zero additional cost, which is one of the advantages of using shaders.

One important thing to mention is that the shader in algorithm 3.1 uses discrete sampling. That is, a pixel is either inside or outside of a matrix cell. It is never half inside or half outside. This property is illustrated in figure 3.2. The example shows that, because of the used sampling method, some matrix cells may end up at different sizes than others. Worse, the same problem manifests inside those cells. For example, figure 3.2c also shows the Barchart visualization when only 9 pixels are available for a single cell. One of the bars is twice as thick as the other, which makes that bar far more visible. The same problem exists for the Squares visualization, where it may happen that the border is not drawn on one side, or simply that the inside has far fewer pixels available than the outside. This is an issue that exists for all methods that require geometry, though the problem may be partly avoided through the use of antialiasing or multisampling.
Algorithm 3.1 GLSL code for the fragment shader which renders the matrix, using the bar chart visualization.

```glsl
1 in vec2 texco; // interpolated texture coordinates (we rendered one quad)
2 // the following (uniform) values are the same for all fragments (pixels)
3 uniform sampler2DRect texture;
4 uniform int matWidth;
5 uniform int matHeight;
6 uniform float maxvalue;
7 uniform float threshold; // between 0 and 1
8 // the matrix cell visualization method itself, in this case: barcharts
9 vec3 getColorInCell(vec2 pos, vec2 sample, float max)
10 {
11 vec3 c = vec3(0,0,0);
12 if(pos.y < log(1.0 + sample.r) / log(1.0 + max) && pos.x < 0.5){
13 c = vec3(0.0, 0.7, 1);
14 }
15 if(pos.y < log(1.0 + sample.g) / log(1.0 + max) && pos.x > 0.5){
16 c = vec3(0.566, 0.7, 1);
17 }
18 return c;
19 }
20 void main() {
21 // conversion to get the indices of the matrix cell that this pixel looks down on
22 vec2 tc = texco * vec2(matWidth, matHeight);
23 float tx = min(float(matWidth -1), floor(tc.x));
24 float ty = min(float(matHeight -1), floor(tc.y));
25 vec2 pos = vec2(tc.x - tx, tc.y - ty);
26 vec2 sample = texture2DRect(texture, tc).xy; // getting the values in the matrix itself
27 sample = max(sample - vec2(max*threshold), vec2(0,0)); // applying the threshold
28 vec3 c = getColorInCell(pos, sample, max*(1.0 - threshold));
29 c.z *= (1.0 - 0.2 * pos.x * pos.y); // apply a gradient
30 c = hsv2rgb(c);
31 gl_FragColor = vec4(c, 1);
32 }
```

Figure 3.2 Discrete sampling. Part (a) shows the matrix cells, with the pixels used to sample shown in blue. Part (b) shows that the pixels are now filled in with values sampled from the matrix. The red box shows the area available to one matrix cell. These pixels are used to display per-cell geometry, such as bar charts (c.1) and boxes (c.2). Note that this geometry is also sampled discretely, so the resulting bar charts may not have the same width.
**Figure 3.3** Spline rendering pipeline. Lines are first rasterized and written to a single channel framebuffer. Then the maximum pixel is computed for both buffers, and the normalized result is rendered.

**Figure 3.4** The effect of the reduction shader on a 4x4 framebuffer. This reduction shader computes the maximum value of four pixels, and writes back the result.

**Non-leaf rendering** This situation does not allow for the use of textures in the same way as in the leaf case. The problem is that matrix cells in non-leaf cases are not necessarily square. That means geometry has to be explicitly generated. The reason for this is that the cell visualizations require information on the position inside a cell. For that, the size of the cell itself is needed. It is desirable that the shader only needs to do a constant number of lookups, so whatever data it receives has to contain information on the location of a cell, its size, and something to compute the fragment position (pos in algorithm 3.1) from. One could use a large texture which contains this information, but that would require the use of a texture of the same size as the leaf level, which would also contain at the very least 4 values per cell. Alternatively, the cells can be explicitly specified, which is significantly faster at higher levels, because there are always relatively few cells at higher levels. The geometry that is generated is put into a vertex-buffer. That buffer contains a large amount of quads, one for each cell in the matrix. Each vertex of each quad has additional attributes, such as the edge strengths $v_1$ and $v_2$. The main disadvantage of this method is that it is significantly slower than the texture-based method. The problem is that each cell takes four vertices, and each vertex needs an x and y coordinate as well as the values $v_1$ and $v_2$. This means each cell requires at least 16 floats, while the texture based method requires only two floats per cell.

The fact that this method uses geometry makes the used shader less complicated. The shader mentioned in algorithm 3.1 remains mostly the same for the non-leaf case, but the whole texture lookup part can be skipped, because the sample and pos variables are embedded in the geometry.

### 3.2. Edge Rendering

Section 2.4 describes a method for rendering many bundled edges. To implement this method, a pipeline as shown in figure 3.3 was used. The difference with the pipeline shown in figure 2.11 is that acquiring the maximum pixel is explicitly shown. Rendering is implemented using framebuffers and OpenGL’s fragment shaders. The fragment shader reads a pixel from the framebuffer for each network, and computes the resulting color.

This method requires the maximum pixel for both framebuffers, one for each network, in order to correctly do a normalization step. The standard approach to getting the maximum pixel would be to transfer the whole framebuffer to RAM, and compute the maximum pixel there. This method
Figure 3.5  Edge rendering using triangle strips. The edge to be rendered is in red. The texture coordinates used by algorithm 3.2 are shown as well. Nodes indicate the borders between line segments that make up the curved line. The final color will be stronger closer to the center, and almost invisible near the edges.

Figure 3.6  Potential issues when using thicker lines combined with sharp corners. When the line width \( w \) is increased to the width shown in blue, the lines perpendicular to the red line strip cross, causing an incorrect result.

The edge bundling method is very dependent on the ability to render lines. However, in OpenGL, lines are generally not anti-aliased. To overcome this issue, we render triangle strips instead. The vertices in the triangle strips have one dimensional texture coordinates, either \(-1\) or \(1\). These values can be generated in the vertex shader using \( \text{gl\_VertexID} \), as shown in algorithm 3.2 which shows the shaders responsible for this process. The fragment shader receives the interpolated result, and computes the final color. The color gets darker the further away from the center the fragment is. This strip will be rendered at a total width of two pixels, but is not aligned to pixel boundaries. Figure 3.5 shows the structure of the triangle strip when rendering the red line strip. It also shows the texture coordinates. Note that this method does not attempt to avoid issues with very sharp corners. Luckily these issues do not manifest in our application, due to our use of thin lines. Figure 3.6 shows why thicker lines can cause problems. Finally, Beziér curves generally avoid sharp corners, which works in our favor as well.

Algorithm 3.2 Vertex and fragment shaders to render antialiased edges.

```c
1  out float tex;
2  3 void main() {
4    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
5    tex = -1.0 + 2.0 * float(gl_VertexID % 2);
6  }

1 uniform vec4 color;
2 in float tex; // interpolated result for this fragment
3 4 void main(){
5    gl_FragColor = color * (1.0 - abs(tex));
6  }
```

The edge bundling method is very dependent on the ability to render lines. However, in OpenGL, lines are generally not anti-aliased. To overcome this issue, we render triangle strips instead. The vertices in the triangle strips have one dimensional texture coordinates, either \(-1\) or \(1\). These values can be generated in the vertex shader using \( \text{gl\_VertexID} \), as shown in algorithm 3.2 which shows the shaders responsible for this process. The fragment shader receives the interpolated result, and computes the final color. The color gets darker the further away from the center the fragment is. This strip will be rendered at a total width of two pixels, but is not aligned to pixel boundaries. Figure 3.5 shows the structure of the triangle strip when rendering the red line strip. It also shows the texture coordinates. Note that this method does not attempt to avoid issues with very sharp corners. Luckily these issues do not manifest in our application, due to our use of thin lines. Figure 3.6 shows why thicker lines can cause problems. Finally, Beziér curves generally avoid sharp corners, which works in our favor as well.
3.3. Interaction

There are 2 selections of tree nodes possible at the same time; these selected nodes are marked in green or gold. Only nodes can be selected, edges can only be highlighted by selecting one of the nodes that edge connects. If both selections are non-empty, only the connections between the selections are shown. If only one selection is non-empty, all edges connected to the selection are shown. Gold and green were chosen because they are visibly very different from the red and blue which indicate the two networks.

3D view  The 3D brain regions are clickable with both mouse buttons. The left mouse button corresponds to the green selection, the right mouse button controls the gold selection. Clicking the same region twice with the left mouse button will clear the green selection. The golden selection can be cleared in the same way. The brain can also be rotated, by dragging with the left mouse button. Lastly, the scroll wheel controls zoom.

2D view  The scroll wheel controls the zoom again. The icicle graphs can be clicked to select a whole subtree. They can also be dragged, which selects a number of subtrees. The left icicle graph corresponds to the golden selection, while the bottom icicle graph corresponds to the green selection. This view can be dragged around by dragging on the matrix or the edge bundling visualization using the left mouse button. Ranges can also be selected by dragging a box in the matrix using the right mouse button. Dragging a box of size $0 \times 0$ will clear the selection. Furthermore, clicking the same region twice in the icicle graph will clear that selection.

Control panel  The zoom slider changes the current hierarchy level to display in the matrix and the 3D view. The matrix render mode dropdown list chooses one of the matrix modes described in section 2.1. The threshold slider changes the threshold as described in section 3.1. The logarithmic checkbox chooses which of the scaling functions from equation 2.1 to use.
4. Results & Discussion

In section 4.1, we will discuss whether or not the questions asked in section 1.4 can be answered using our tool. After that, we will validate whether experts agree with our observations in section 4.2. We will also validate whether or not the questions asked in section 1.4 are actually relevant, and whether or not that list is exhaustive. Finally, we will make some observations on the scalability of our application.

4.1. Results

Most of the questions mentioned in section 1.4 can be answered using our application.

- To what extent are two regions connected. Does this connection differ between the two networks?

A connection between two regions is a cell in the adjacency matrix, as well as a line in the 3D view. The absolute value of the connection in both graphs can be seen in the control panel when hovering over the corresponding cell in the matrix. Figure 4.1 shows that the control panel contains information about the connection that is being hovered over. This information can be found in the highlighted node block.
• How connected is a region? Is there a significant difference between the two networks? Maybe the second network has fewer, but stronger connections?

The connectivity of a region is a column in the adjacency matrix. Such a column is selected in figure 4.2, and the total number of connections that column contains is shown in the control panel. However, the total strength of these connections is not shown. Cells in the selected column can be compared visually, but can also be hovered over. When hovered, the connection strength for that cell is shown in the control panel, underlined in black.

• Where are hubs, and do they have the same size for both networks? Are they the same otherwise?

Hubs show up as blocks in the matrix due to the used sorting algorithm, as shown in figure 4.3. Furthermore, the edge bundling visualization shows hubs as a large group of edges that connect neighboring nodes. Figure 4.1 shows that, due to the use of logarithmic scaling, the matrix does not clearly show hubs. The edge bundling visualization, however, does show hubs and strong bundles, due to its different use of logarithmic scaling.

• Which other region does this region mainly connect to? Is it the same region for both networks?

This can be solved by finding the column in the matrix corresponding to the given region, as shown in figure 4.2, and finding the brightest cell. Disabling logarithmic scale may help, along with modifying the threshold. This should narrow down the search to only a few regions, which can then be compared by looking at the absolute values in the control panel. Any differences are easily seen in the adjacency matrix because of the use of colors, and are also shown in the control panel. This can also be solved using the 3D anatomical view, by selecting the region, which will show all outgoing edges. Changing the threshold should allow the user to identify the strongest connection as well, though this method does not show the absolute values in the control panel.

• Where, anatomically, are the differences located?

After finding interesting differences, the regions in question can be selected, and will be highlighted in the 3D view. Furthermore, the relevant edges will also be drawn there. When hovering over a cell in the matrix, the regions that are on the X and Y axis will be highlighted in 3D using an outline. When regions are selected, they will be the only colored regions. In figure 4.4, the cell at \((x, y)\) is hovered over, causing the white outline, and the nodes \(x\) and \(y\) are both selected as well, giving them their color.

• Which regions are the most different, and which regions are identical?

This question cannot be answered easily. A region that is very similar in the two networks can be seen as a column that is almost entirely monochrome, when using the saturation method. Finding the most different connection, can be answered using the diff method. The brightest cell corresponds to the largest difference, though it may be difficult to identify the brightest cell when one bright cell is blue and the other red. If only a handful of regions remain due to thresholding, hovering over those connections in the matrix shows the absolute values, but this is of course not an option if many regions have a large difference.

### 4.2. Evaluation

We held two evaluation sessions, the first session had two experts, the second session one. The first two participants were neuroscientists that analyzed the presented dataset before. The second session had a participant who works on developing solutions for brain connectivity analysis, and also knew the presented dataset well. The fact that all participants already knew the datasets allowed them to more easily identify regions of interest, and made the evaluation sessions go smoothly.

The evaluation process was split into three parts, first we analyzed whether or not the questions asked in section 1.4 were relevant for the users that were participating. Then, the user was guided
through the tool, and was allowed to explore his own datasets. The user was subsequently asked to execute some tasks, and indicate if the tool was well suited for the given task. Finally, the user had to answer a few open questions. This was all done through the use of a questionnaire, which can be found in appendix E. Both sessions took about one hour.

### 4.2.1. Task Analysis

The table below contains a number of statements whose importance needed to be verified. The importance is rated according to the Likert [Lik32] scale:

1. Strongly disagree 2. Disagree 3. Neither agree nor disagree
4. Agree 5. Strongly agree

In the table below, the response for each participant is shown. The participants from the first session are shown in red and gray. The participant from the second session is shown in blue.

<table>
<thead>
<tr>
<th>General network analysis</th>
<th>Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>It is important to be able to ...</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>1.1. ... see how strong the connection between two regions is.</td>
<td>![red] ![gray]</td>
</tr>
<tr>
<td>1.2. ... see how many other regions a given region is connected to.</td>
<td>![red] ![gray]</td>
</tr>
<tr>
<td>1.3. ... find strongly connected components.</td>
<td>![red] ![blue]</td>
</tr>
<tr>
<td>1.4. ... find which region a given region mainly connects to.</td>
<td>![red] ![gray]</td>
</tr>
<tr>
<td>1.5. ... see if the connection strength between two regions is different.</td>
<td>![red] ![gray]</td>
</tr>
<tr>
<td>1.6. ... see how much the connection strength between two regions differ.</td>
<td>![red] ![gray]</td>
</tr>
<tr>
<td>1.7. ... determine if a given region has fewer outgoing connections in one network than in the other.</td>
<td>![red] ![gray]</td>
</tr>
<tr>
<td>1.8. ... determine if a given region has more but weaker outgoing connections in one network than in the other.</td>
<td>![red] ![gray]</td>
</tr>
<tr>
<td>1.9. ... see if one network has a strongly connected component while the other does not.</td>
<td>![red] ![gray]</td>
</tr>
<tr>
<td>1.10. ... see if one region’s main peer is the same for both networks.</td>
<td>![red] ![gray]</td>
</tr>
<tr>
<td>1.11. ... find the anatomical location of the regions that are most different.</td>
<td>![red] ![gray]</td>
</tr>
<tr>
<td>1.12. ... find the regions that are the most different, regarding their connections with other regions.</td>
<td>![red] ![gray]</td>
</tr>
<tr>
<td>1.13. ... find the regions that are the most similar, regarding their connections with other regions.</td>
<td>![red] ![gray]</td>
</tr>
</tbody>
</table>

The above table shows that the tasks set in the beginning of this thesis were appropriate for these users. The biggest difference between the experts seems to be the interest in outliers, as evidenced by questions 1.12 and 1.13.

We asked whether or not there were any tasks that should have been included in the list above. One of the experts mentioned that it would be nice to have an export function that exports all information on the currently selected connections or nodes. This would allow our tool to be used for an initial exploration of the data, after which other tools can take over.

### 4.2.2. Tasks

In this part of the evaluation, we asked the users to perform tasks, during which we recorded their accuracy, results, and remarks. The tasks the users had to perform were chosen to validate whether the tool is capable of performing the tasks mentioned in section 4.2.1. The exact tasks that were
used are detailed in appendix E. Note that the two participants in the first session did these tasks together, so they only provided one answer.

Before attempting any of these tasks, we gave a small demo to showcase the features of our application. The focus was on interaction, so the participants would be able to perform the task below. While doing the tasks, participants were reminded of the features the tool has where appropriate.

2.1. This task consisted of two parts. The first part was to find a brain region, given the name (lh.frontalpole). This was accomplished very quickly by all users using the 3D visualization. The second part was to look at the total connection strength. The user was asked to decide which network had a greater total connection strength, after which he thought about computing the sum. However, this question could easily be answered by looking at the few largest values. The total connection strength of a node is interesting information, which was included in the future work section. When asked how suitable the tool was to answer this question, the result was a 4 on the Likert scale. During the second user evaluation, with a different user, it was mentioned that it would be more intuitive to show highlight information using a tooltip, instead of showing it in the control panel. The tool did, however, receive a 5 on the Likert scale during the second evaluation.

2.2. The second task is about finding which region mainly connects to a given region. The users quickly solved this problem using the threshold feature and by swapping between the red only and blue only matrix modes. The tool received a 4 and a 5 on the Likert scale for this task.

2.3. This task was about finding two regions, and getting the absolute connection strength. This was easily accomplished by clicking on the regions in 3D, and selecting both regions. Hovering over the selected cell in the matrix shows the actual values. The tool received a 5 on the Likert scale. During the second evaluation, it was mentioned that the control panel should not only show the connection strength belonging to each network, but also the difference between the two. This is implemented in the final version of the tool. Again, the tool received a 5 on the Likert scale during this evaluation.

2.4. Task four was about getting the total number of connections to one region. Selecting the region shows the number of connections in the control panel. The tool twice received a five out of five on the Likert scale.

2.5. The last task was about finding the strongest connection in the entire network, and then finding the anatomical location of the regions involved. This was solved by adjusting the threshold all the way up, so that only the strongest connection remains. Selecting that connection highlights both regions in 3D, which answers the question. The tool received a five on the Likert scale, during both evaluation sessions.

4.2.3. Open Questions

The open questions revealed that the aspect of our visualization that adds the most to the user’s current workflow is that all visualizations are integrated, selecting a region in one visualization highlights that region in the other. The users also expressed a desire to have more visualizations, of other data. An example of a desired additional visualization is a streamline [Tuc02] visualization. The network data we focused on is a very abstract representation of the data, and is very useful for finding interesting properties. When such a property is found, the user wants to be able to dive down to the data from which the network was generated, seeing the relevant parts highlighted.

The fact that the tool is able to compare two networks is very useful. It allows comparing two groups of people, the same person at different points in time, or maybe simply a differently thresholded connection matrix. Having the ability to compare two networks allows the user to create filter networks too, which simply contain only the part that is or is not interesting, which would make the interesting parts stand out.
Matrix visualization

The main disadvantage that was mentioned is that region names are not shown in the borders. However, they are available by hovering, and are not even all that interesting when using a random parcellation of 1000 nodes, because most regions in such a random parcellation do not have a useful name.

During the evaluation, we asked the participants to rank the cell visualizations, in section 2.1, best to worst. During the first session, the first place went to the saturation encoding, second to the diff encoding, third to diff squares, and squares, barcharts and triangles shared the fourth place. During the second user evaluation, the first three spots go to saturation, diff, and diff squares again. The fourth place goes to bar charts, and the fifth to the squares method. Finally the triangles method takes the last place. This ordering of cell visualizations is also shown in figure 4.5. Surprisingly, these results do not correspond with the conclusions by Alper et al. \cite{ABHR13}. They conclude that the squares method is leagues ahead of bar charts and triangles, though of course their tests do not include our methods: saturation, diff, and diff squares. It is therefore possible that the squares method is simply outclassed by, for example, saturation, which would mean there is no reason to ever use squares, whereas the bar chart method could still be useful when zoomed in, due to its clarity and simplicity. Another important note is that Alper et al. focused on small matrices, while we focus on large matrices. Focusing on large matrices puts the methods by Alper et al. at a large disadvantage, due to their use of geometry, which helps explain why their methods appear inferior. Finally, our test only had three participants, of which two influenced eachother, so our results are based on a very small sample size.

In general though, the matrix visualization was very similar to what the experts used already.

Edge bundling visualization

None of the tasks were formulated in such a way that the edge bundling visualization would be required. The result was that this visualization was mostly ignored by the participants, so we will refrain from commenting on its effectiveness.

3D anatomical view

The 3D anatomical view works in the user’s preferred space, so it is very familiar. It shows a lot of the desired information, in a very accessible way. The way edges are rendered here is not something the users saw before. Especially the fact that it is interactive makes the tool special. It is, however, a bit difficult to work with sometimes, the lines are thin, can cause clutter, and take a long time to render. Encoding the edge information in the color of the target region for each edge would be a potential improvement. However, this may be difficult to implement for regions that have many incoming edges, because it can be difficult to decide what color should be assigned to those regions.

Missing features that might be interesting for future work are the links with streamlines \cite{Tuc02}, fibers, and volume data. When selecting a region, and seeing the connections from and to that region, it would be interesting to see the streamlines that led to each connection. Furthermore, some of the volume data that was used to generate the streamlines could be visualized too, linked with the current visualizations so areas are highlighted appropriately. Some of these visualizations are already included in the framework we used, vIST/e \cite{Ein15}, but are currently not linked to our visualizations.
4.3. Scalability

The scalability of our program is mainly decided by the amount of memory it uses. The memory usage for the following components scale quadratically with the number of nodes the graph has:

- Edge bundling. The running time for this visualization also scales linearly with the number of edges.
- Matrix visualization.

The memory usage of the 3D visualization scales linearly with the number of vertices in the 3D mesh. Furthermore, it scales linearly with the number of edges selected. The time required to render one line scales with a factor $O(\log(n))$ where $n$ is the number of vertices in the 3D mesh, because of surface springs.

One extra limitation is that the memory used by the edge bundling visualization and the 3D visualization must be contiguous, and must fit in graphics memory.

Lastly, the matrix ordering algorithm requires $O(v^3)$ time to compute a distance matrix for the Tanimoto distance, see section 2.2. Here $v$ is the largest number of children any node in the hierarchy has.

A dataset with 1000 nodes and a mesh of 200000 triangles was tested on a machine with an AMD E-350 CPU (2x 1.6GHz), 6GB of RAM, and an AMD Radeon HD 6310 GPU. At these sizes the application was still usable at interactive framerates, and was not noticeably slower than it was with the same mesh and 500 nodes.

The program also scales with screen size. The 3D mesh is rendered to a framebuffer, potentially the size of the screen. Furthermore, the program makes use of fragment shaders. A larger screen resolution means more pixels, and more fragments. A larger screen resolution will influence the performance of the application. However, during our testing there was no significant performance difference between using 1366 $\times$ 768 or 1920 $\times$ 1080.
5. Conclusions

In this thesis we discussed ways to visualize differences in brain networks. We designed an application capable of visualizing brain networks in general, with a focus on highlighting differences between two networks.

In sections 4.1 and 4.2 we showed that our tool is capable of answering the key questions posed in section 1.4. Section 4.1 explains how our application can be used to answer those key questions. Section 4.2 details our user evaluation, in which experts confirm that our application indeed answers the key questions asked in section 1.4. It is also shown that the experts generally like our application, and believe it capable of answering the questions asked. They do have suggestions, but many of those are not covered by our key questions, and are therefore outside the scope of this project. These suggestions are, however, mentioned in the future work section.

Our application is demonstrated to be able to find the extent to which two regions are connected, and confirm if a connection differs between the two input networks. It is also able to display the total number of connections a region has, but does not show the total connection strength. Strongly connected components can also be identified, and selected regions are shown in 3D. This means the requirements set in the beginning of this thesis have been met.

We conclude that our application provides new insight into brain networks, and consequently the workings of the brain. Furthermore, we believe that this project can be used as a starting point for future research into visualization of differences between two brain networks, and brain network visualization in general. We discussed that many aspects of the problem still need further research, but this project is a step in the right direction.

5.1. Future Work

Our solutions all focus on differences in edges between nodes, but ignore all per-node information. It would be interesting to visualize per-node information alongside the current visualizations. Showing per-node information also allows the user to easily identify the most similar region or most different region between networks, as well as visualizing the total connection strength of each node. Per-node information is not a new thing, this is something the experts already have. However, their implementation is not integrated with our visualizations, which can be very useful.

During the user evaluation, it was mentioned that it would be useful to use another color scale. Right now, we mostly use a scale from red to white to blue, but it may be interesting to split the red and blue parts into two different colors. Right now, the value is encoded in the brightness, but the range of values is very large and it can be difficult to discern between two slightly different values. A color scale that differs more could be a solution. However, we later held another user evaluation, where we proposed the use of such a color scale. The immediate response was that this would negatively impact the intuitiveness of the visualization. This makes the use of an alternate color map an interesting issue, and worthy of further study.

The 3D visualization we implemented uses lines to render connections between regions on the surface. An alternate method to render these connections is by coloring the surface patches themselves. The main advantage of this method is that it results in less clutter, but the main disadvantage is that it may be hard to encode the same amount of information. Instead of replacing the lines, colored surface patches could also be used to convey the node statistics mentioned earlier.

Another issue with the 3D visualization is that it currently cannot show subcortical areas. These areas would be located inside the cortical mesh (the mesh we are currently rendering), which conflicts with the requirement that our 3D lines never clip through the surface. A solution, such as moving the subcortical area next to the cortical areas, would have to be used.

Another issue with the 3D visualization is that it currently cannot show subcortical areas. These areas would be located inside the cortical mesh (the mesh we are currently rendering), which conflicts with the requirement that our 3D lines never clip through the surface. A solution, such as moving the subcortical area next to the cortical areas, would have to be used.

During the user evaluation, it was mentioned that it would be desirable to see additional information. The network data we received is the result of a lot of processing. This makes it relatively easy to work with, but it is, by nature, a simplification of the actually measured data. This makes it
interesting to look at the original data, after finding regions of interest using our tool. Ideally, those regions of interest would be highlighted in the lower-level datasets too. For example, the data from Maastricht was built by finding streamlines in a dataset measured by an MRI scanner, and then aggregating these streamlines to form a network. This means that, for this dataset, there are two lower-level data structures of interest. It would be interesting to look at the streamlines that generated a connection, or even to look at the original volume data that the streamlines were obtained from.

When designing our application, the assumption was made that hierarchies match between networks. That is, the nodes themselves match one on one, and they have exactly the same hierarchical structure. The datasets we obtained from Maastricht were randomly parcellated, so they cannot be compared by our application. Coming up with a suitable matching between nodes can be an adequate solution, but this would likely be best solved differently. For example, by building a visualization that does not require a one on one matching between nodes in both networks.

A feature request that surfaced during a meeting with experts is pre-selection. The idea here is that the user selects a few nodes, and all other nodes in the network are collapsed to one node which represents the rest of the network. This allows the user to focus on small parts of the network, without being overwhelmed by the total network. We implemented this feature in the preprocessor, see section 2.6, but this is a command line tool. Ideally, this feature would be accessible interactively. This would, however, be a change in the network structure and size, while the application was developed under the assumption that the network remains constant.

Currently, clusters of similar nodes are shown in the adjacency matrix visualization, but that visualization is limited to showing clusters with a linear ordering. This limitation makes the adjacency matrix a suboptimal method for displaying clusters. Furthermore, we do not explicitly look for clusters when ordering the matrix. Therefore, we believe that the task of identifying clusters would best be solved using a different visualization.
6. References


Appendices

A. Cocomac File Format

The cocomac file format is based on data from the CoCoMac (http://www.cocomac.org/) project, but has been modified to contain hierarchical information.

An example cocomac file:

```
# a comment
0,Node name, x, y, z, A1
1,Node name2, x2, y2, z2, A2
2,Node name3, x3, y3, z3, PCM
3,Parent 1 --> 0,1
4,Parent 2 --> 2
5,Root node --> 3,4

, 1, 2
0, , 1
3, 1,
```

All lines until the first empty line describe nodes. Non-leaf nodes are described using their ID, a name, and an arrow (--> ) followed by the indices of all children. Leaf definitions do not have the arrow, but instead define 3d coordinates as well as a short name like “A1”.

After the first empty line, the matrix definition starts. The matrix is always square, and its width is equal to the number of leaves defined earlier. Cells can contain the following values:

- empty: No information available
- 0: No connection
- 1: Weak connection
- 2: Moderate connection
- 3: Strong connection
- X: Connection of unknown strength
## B. Barcelona File Format

The Barcelona file format is a python `networkx` object. `networkx` is a library for python, and is meant for analyzing graphs. The python `pickle` library is used to serialize the python object.

The `gpickle` files supplied by Barcelona can be loaded directly by the `pickle` library, after which a normal `networkx` object remains.

For more information, see the following urls.

- [https://docs.python.org/3.4/library/pickle.html](https://docs.python.org/3.4/library/pickle.html)
- [https://networkx.github.io/](https://networkx.github.io/)

The barcelona data does not contain a 3D mesh, which required the use of an external brain mesh. To map nodes to the 3D mesh, we used the following method.

The XHGFZ format specifies the associated node per vertex, so we need our labels per vertex, not per face. We normalized coordinates of all vertices in the mesh first, such that the coordinates are in the range $-1..1$, while preserving aspect ratio. The result is, for example, that the $x$ direction has range $-1..1$, but the $y$ direction has range $0.8..0.8$. Then, we independently normalized the 3D locations specified by the `networkx` object, such that they are in the same range as the vertices from the mesh. Then, every vertex is simply assigned to the nearest node.

Later, we received a matching brain mesh and associated labels. The brain meshes are `srf` files, an example of which is shown below. These files start with a comment. Line two describes how many vertices and how many faces there are. In the example below, there are 121582 vertices. The following lines are either vertex definitions or face definitions. A vertex consists of x, y, z and 0. A face consists of three indices into the vertex list, and a zero.

```ascii
#! ascii version of rh. pial
121582 243160
12.437897 -81.744690 -14.103851 0
11.779770 -81.830399 -13.941169 0
10.493901 -81.983719 -13.822988 0
9.257098 -82.017494 -13.781264 0
8.490166 -81.982819 -14.196484 0
7.802504 -81.788315 -13.923920 0
... 
0 1 6 0
7 6 1 0
0 60 1 0
61 1 60 0
0 69 60 0
0 69 60 0
1 2 7 0
8 7 2 0
```

Each vertex is assigned to a node. The data for this is in a zip file. The structure of that zip file is shown in figure B.1.

The `.label` files have the following structure:

```ascii
#! ascii label, from subject FREESURFER vox2ras=TrReg
1235
36068 48.457 -34.000 -4.788 0.000000000
36080 49.139 -34.381 -5.655 0.000000000
36081 48.201 -34.247 -5.382 0.000000000
36082 47.626 -34.268 -5.559 0.000000000
36090 51.423 -34.725 -6.875 0.000000000
36091 50.494 -34.572 -6.645 0.000000000
36092 49.509 -34.342 -6.538 0.000000000
```

The first line is a comment, and the second line describes how many vertices this file contains. Every vertex definition has 5 elements: an index, xyz coordinates and a zero. We only use the index, which is an index into the vertex list described by the `.srf` file. In other words, the zip file corresponds to a mapping from node name to index. Some names that are mentioned in this file do not exist in the network, so these labels are ignored. After assigning the correct label to each vertex, a flood fill algorithm is used to make sure unlabeled vertices do receive a label.
Figure B.1  The structure of the vertex label zip file.
C. Maastricht File Format

The Maastricht data is partitioned in 3 files which provide metadata, and two files per dataset which provide the adjacency matrices.

C.1. Info File 1

This file is a .mat (matlab workspace) file, which contains two interesting matrices. These matrices describe the hierarchy for this network.

- refinedNewMap
- newMap100

The maastricht data has been obtained using a parcellation of size 100000. This is then clustered using an algorithm, down to sizes 4000, 2000, 1000, 500, and 100.

The two matrices, refinedNewMap and newMap100 contain more or less the same information. In the table below, the column “100 parcel” can be found in newMap100, while the other columns can be found in refinedNewMap.

<table>
<thead>
<tr>
<th>index</th>
<th>100 parcel</th>
<th>500 parcel</th>
<th>1000 parcel</th>
<th>2000 parcel</th>
<th>4000 parcel</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>100000</td>
<td>100</td>
<td>500</td>
<td>1000</td>
<td>2000</td>
<td>4000</td>
</tr>
</tbody>
</table>

These two matrices assign each parcel from the 100000 parcellation to a parcel in one of the smaller parcellations. The example shows that, for example, for the 100 and 500 parcellation the 10th parcel consists of 4 parcels from the 100000 parcellation. It also shows that the 1000 parcellation splits parcel 10 in two pieces, 10 and 11. The 2000 parcellation splits 11 up further, in 11 and 12.

More formally, this matrix describes the following function:

\[ [1 \ldots 100000] \rightarrow [1 \ldots 100] \times [1 \ldots 500] \times [1 \ldots 1000] \times [1 \ldots 2000] \times [1 \ldots 4000] \]

C.2. Info File 2

This file is a .mat (matlab workspace) file, which contains two matrices of interest. These matrices describe the xyz coordinates for each node in the 100000 parcellation, as well as a mapping for each node in the 100000 parcellation to a name, provided in a csv file described in section C.3.

- coordinates
- atlasToSrf
Coordinates is a matrix, 100000 elements long, and 3 elements wide.

<table>
<thead>
<tr>
<th>index</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1</td>
<td>2</td>
<td>-0.2</td>
</tr>
<tr>
<td>2</td>
<td>0.6</td>
<td>1.2</td>
<td>-0.7</td>
</tr>
<tr>
<td>3</td>
<td>0.4</td>
<td>1.5</td>
<td>-0.8</td>
</tr>
<tr>
<td>4</td>
<td>0.9</td>
<td>1.3</td>
<td>-0.4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>100000</td>
<td>0.3</td>
<td>-0.1</td>
<td>0.1</td>
</tr>
</tbody>
</table>

More formally, this corresponds to the following function:

$$[1 \ldots 100000] \rightarrow \mathbb{R} \times \mathbb{R} \times \mathbb{R}$$

atlasToSrf is a matrix, 100000 elements wide, and 1 long. It provides an index into a csv file which contains names. For example:

<table>
<thead>
<tr>
<th>node1</th>
<th>node2</th>
<th>\cdots</th>
<th>node100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>\cdots</td>
<td>164</td>
</tr>
</tbody>
</table>

These indices correspond to indices in Infofile3, where each of these indices is assigned a name. More formally, this corresponds to the following function:

$$[1 \ldots 100000] \rightarrow [2 \ldots 164]$$

### C.3. Info File 3

This file is a .csv file, which contains a list of names and indices.

| niftiValue;areaCaret;oldCocomacAcronym;areaCocomac;areaName;hemisphere;finalAssignment |
| RM_TCpol;R00 - TCpol;RM - TCpol;temporal polar cortex;right;RM - TCpol_R |
| RM_TCs;R00 - TCs;RM - TCs;superior temporal cortex;right;RM - TCs_R |

### C.4. Data files

There are several available adjacency matrices, sized 100, 500, 1000, 2000, and 4000. All of those are .mat files, containing one variable, “adjMatrix”.

This matrix is a float array, an example for the 100 parcellation can be seen below.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>\cdots</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.01</td>
<td>2</td>
<td>3.1</td>
<td>\cdots</td>
</tr>
<tr>
<td>2</td>
<td>0.4</td>
<td>5.1</td>
<td>1.1</td>
<td>\cdots</td>
</tr>
<tr>
<td>3</td>
<td>0.6</td>
<td>6.7</td>
<td>3.2</td>
<td>\cdots</td>
</tr>
<tr>
<td>...</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>100</td>
<td>0.3</td>
<td>1.41</td>
<td>2.1</td>
<td>0.8</td>
</tr>
</tbody>
</table>

More formally, this corresponds to the following function:

$$[1 \ldots N] \times [1 \ldots N] \rightarrow \mathbb{R}^+$$

for an adjacency matrix of size $N \times N$. 

59
D. XHGFZ File Format

XHGFZ stands for X Hierarchical Graph Format gZipped. This appendix describes version 3. An XHGFZ file contains several levels, each containing an adjacency matrix and a number of nodes. Level 0 is the lowest level, the one with all the leaves. Higher levels contain pointers to lower levels to indicate children nodes.

XHGFZ is a gzipped file format which contains a tree structure. After g-unzipping, the general structure is as follows:

- block id: byte
- length: 4 bytes signed int (big endian)
- data: nr of bytes specified by length

D.1. Block IDs

1. nodelist
2. node
3. matrix
4. string
5. version
6. layer
7. nodeName
8. geometry
9. vertexList
10. faceList
11. nodeMap

D.2. Block Descriptions

These descriptions describe the “data” attribute of the block.

**Nodelist** contains node blocks

**Node** contains the following data (all big endian):
- index: int. This describes the index in the layer below this one where its children nodes are described.
- size: int. The number of children. ‘Index’ should be ignored if this is 0.
- parent: the id of the parent. This is an index in the layer above this one. -1 if there is no parent.
- dummy: byte (0 or 1). See Dummy nodes.

**Matrix** a 2d matrix of 32 bit integers. dimensions are nodelist.length * nodelist.length. This should match blocksize / 4.

**String** The bytes for a string in utf8. Not zero terminated, use the length attribute of the block.

**Version** Contains one integer. Since this appendix describes version 3, the integer must be 3.

**Layer** contains: nodelist, matrix blocks, nodeMap blocks.

**Node name** The same as string. Contains the name of the node defined before this block. Only appears in nodelist blocks, following node blocks. This block is optional, not all nodes will have a nodeName block associated with it.

**Geometry** Contains exactly one “Vertex” List block and exactly one “Face List” block.

**Vertex List** Contains vertices. The number of vertices \( n = \frac{b}{3 \times 4} \), where \( b \) is the size of this block, in bytes. Every vertex is described by 3 floats, each 4 bytes. So the data is something like \( x_1, y_1, z_1, x_2, y_2, z_2 \) etc.
**Face List** Contains vertex indices. The number of faces $n = b/(3 \times 4)$, where $b$ is the size of this block, in bytes. Every face is described by 3 integers, each 4 bytes. A face defined by $f_1, f_2, f_3$ corresponds to the face described by vertices[$f_1$], vertices[$f_2$], vertices[$f_3$], where vertices[i] $\in \mathbb{R} \times \mathbb{R} \times \mathbb{R}$.

**Node Map** Contains node indices. The number of indices should equal the number of vertices defined in the Geometry block. Every index in this block corresponds to a node in the current level. So in a situation with 100 vertices and 10 nodes, this block will contain 100 indices, each in the range 0 . . . 9.

In other words, this block describes which vertices belong to which node.

## D.3. Dummy Nodes

Some nodes are marked as dummy nodes. The hierarchy described in this file always has the property that all leaves have the same depth. In case the original graph from which this file was generated did not have that property, dummy nodes will be added. These nodes are marked as dummy.

## D.4. Tree Representation

Tree of a standard file with 3 layers:

```
version
string
layer 0
    nodelist
        node
            nodeName
            node
                nodeName
                node
                    nodeName
                    matrix
                        nodeMap
                layer 1
                    nodelist
                        node
                            nodeName
                            node
                                nodeName
                                node
                                    nodeName
                                    matrix
                                        nodeMap
                layer 2
                    nodelist
                        node
                            matrix
```
This file format uses the concept of layers. It describes a graph, with a hierarchy. An example hierarchy is given below:

```
layer 0
  d
  e
  f
layer 1
  b
  c
layer 2
  a
```

Then each layer has a matrix, such as:

```
<table>
<thead>
<tr>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>f</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>0.5</td>
</tr>
<tr>
<td>c</td>
<td>0.5</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
</tbody>
</table>
```

Each of these sub matrices describe the connections between nodes at that layer.

Nodes also have an index variable. In the above example, the index variables are as follows:

- a: index = 0, size = 2, parent = -1
- b: index = 0, size = 2, parent = 0
- c: index = 2, size = 1, parent = 0
- d, e: index = -1, size = 0, parent = 0
- f: index = -1, size = 0, parent = 1

So the index and range variables describe a range in the lower layer which contains its children.

Lastly, layers can have a nodeMap. The nodeMap describes which vertices belong to which node. Example node maps for a surface with 10 vertices is given below. Note that a nodeMap is mandatory for every layer except for the root, when a geometry block is specified. The nodemap contains indices, which indicate which node is meant. In this case, at level 0, the nodelist is \([d, e, f]\) and at level 1 the nodelist is \([b, c]\).

Level 0 nodeMap:

```
<table>
<thead>
<tr>
<th>v1</th>
<th>v2</th>
<th>v3</th>
<th>v4</th>
<th>v5</th>
<th>v6</th>
<th>v7</th>
<th>v8</th>
<th>v9</th>
<th>v10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
```

Level 1 nodeMap:

```
<table>
<thead>
<tr>
<th>v1</th>
<th>v2</th>
<th>v3</th>
<th>v4</th>
<th>v5</th>
<th>v6</th>
<th>v7</th>
<th>v8</th>
<th>v9</th>
<th>v10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
```
E. Questionnaire

E.1. Questions

To determine which features the application should support, we used a number of questions as a starting point. These questions describe what we believe you want to extract from the data, and we would like to verify that the questions we used are indeed important.

For the following statements, please rate their importance according to the following scale:


It is important to be able to ...

<table>
<thead>
<tr>
<th>General network analysis</th>
<th>Important</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1. ... see how strong the connection between two regions is.</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>1.2. ... see how many other regions a given region is connected to.</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>1.3. ... find strongly connected components.</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>1.4. ... find which region a given region mainly connects to.</td>
<td>1 2 3 4 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Network difference focused</th>
<th>Important</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.5. ... see if the connection strength between two regions is different.</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>1.6. ... see how much the connection strength between two regions differ.</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>1.7. ... determine if a given region has fewer outgoing connections in one network than in the other.</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>1.8. ... determine if a given region has more but weaker outgoing connections in one network than in the other.</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>1.9. ... see if one network has a strongly connected component while the other does not.</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>1.10. ... see if one region’s main peer is the same for both networks.</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>1.11. ... find the anatomical location of the regions that are most different.</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>1.12. ... find the regions that are the most different, regarding their connections with other regions.</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>1.13. ... find the regions that are the most similar, regarding their connections with other regions.</td>
<td>1 2 3 4 5</td>
</tr>
</tbody>
</table>

1.14. Are there any tasks that were not listed but should have been? If so, please write them below.
Demo & Exploration

First you will get a small demo, after which you should explore the data and get a bit more familiar with the tool.

E.2. Tasks

Please answer the questions below. Also rate whether or not you think the tool is well suited for answering each question.

<table>
<thead>
<tr>
<th>Task</th>
<th>Well Suited</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1. In the control network, lh.frontalpole has far more outgoing connections than in the MS network. Is the total connection strength for the MS network smaller? Similar? Stronger?</td>
<td><img src="rating1" alt="Rating" /> <img src="rating2" alt="Rating" /> <img src="rating3" alt="Rating" /> <img src="rating4" alt="Rating" /> <img src="rating5" alt="Rating" /></td>
</tr>
<tr>
<td>Answer:</td>
<td></td>
</tr>
<tr>
<td>Remarks:</td>
<td></td>
</tr>
<tr>
<td>2.2. Which region mainly connects to rh.precentral? Is it the same for both networks?</td>
<td><img src="rating1" alt="Rating" /> <img src="rating2" alt="Rating" /> <img src="rating3" alt="Rating" /> <img src="rating4" alt="Rating" /> <img src="rating5" alt="Rating" /></td>
</tr>
<tr>
<td>Answer:</td>
<td></td>
</tr>
<tr>
<td>Remarks:</td>
<td></td>
</tr>
<tr>
<td>2.3. How strong is the connection between rh.frontalpole and lh.frontalpole? Is it very different between the two networks?</td>
<td><img src="rating1" alt="Rating" /> <img src="rating2" alt="Rating" /> <img src="rating3" alt="Rating" /> <img src="rating4" alt="Rating" /> <img src="rating5" alt="Rating" /></td>
</tr>
<tr>
<td>Answer:</td>
<td></td>
</tr>
<tr>
<td>Remarks:</td>
<td></td>
</tr>
<tr>
<td>2.4. How many regions are connected to rh.frontalpole in each network?</td>
<td><img src="rating1" alt="Rating" /> <img src="rating2" alt="Rating" /> <img src="rating3" alt="Rating" /> <img src="rating4" alt="Rating" /> <img src="rating5" alt="Rating" /></td>
</tr>
<tr>
<td>Answer:</td>
<td></td>
</tr>
<tr>
<td>Remarks:</td>
<td></td>
</tr>
<tr>
<td>2.5 Find the strongest connection, and find the two regions it connects. Where are they in the 3D view?</td>
<td><img src="rating1" alt="Rating" /> <img src="rating2" alt="Rating" /> <img src="rating3" alt="Rating" /> <img src="rating4" alt="Rating" /> <img src="rating5" alt="Rating" /></td>
</tr>
<tr>
<td>Answer:</td>
<td></td>
</tr>
<tr>
<td>Remarks:</td>
<td></td>
</tr>
</tbody>
</table>
E.3. Open Questions

3.1. Which visualization adds the most to your current workflow and why?

3.2. What is your general impression of the tool?

3.3. Which visualization differs the most from what you currently use, and how do you value this visualization?

3.4. What are the main advantages and disadvantages of the Matrix visualization?

3.5. What are the main advantages and disadvantages of the Edge bundling visualization?

3.6. What are the main advantages and disadvantages of the 3D visualization?

3.7. Rank the matrix visualizations based on your preference. Assign the numbers 1, 2, 3, 4, 5, 6 to the following options (lower is better). If you believe two options are more or less equal, just use the same number as the other. For every option, please explain the rank you gave it.

   ___ Saturation  ___ Boxes  ___ Diff boxes  ___ Diff  ___ Bar charts  ___ Triangles