University of Victoria
Faculty of Engineering
Spring 2012 Work Term Report

SVG-Based Visualization of Biomedical Ontology Data

The Chisel Group
University of Victoria
Victoria, British Columbia

David Rusk
V00662179
Work Term 4
Computer Engineering
drusk@uvic.ca

April 28, 2012

In partial fulfillment of the requirements of the
B.Eng. Degree

4222 Thornhill Cres.
Victoria, British Columbia
V8N 3G5

Duncan Hogg
Co-operative Education Coordinator
Department of Engineering Co-op
University of Victoria
Engineering & CSc/Math Co-op & Career Services
P.O. Box 3055
Victoria, BC, V8W 3P6, Canada

April 28, 2012

Dear Mr. Hogg,

Please find enclosed my work term report entitled "SVG-Based Visualization of Biomedical Ontology Data."

This report documents one of the tasks I completed while working for the CHISEL Group at the University of Victoria for my fourth co-op work term. This work term was done after I completed term 3B of the Computer Engineering program.

The CHISEL Group studies computer human interactions in a variety of contexts. The goal of the project I was working on is to provide flexible visualization of biomedical concepts and how they are related and mapped to each other. The main way in which this is done is through a graph view, where concepts are represented as nodes and relationships as arcs. A Flash-based implementation already existed, but one of the goals of the project was to become compliant with open standards. Therefore, I developed a replacement implementation using SVG.

I would like to thank my supervisor Lars Grammel for his guidance and mentoring throughout this work term.

Sincerely,

David Rusk

Table of Contents

List of Figures iii
Summary iv
Glossary v
1.0 Introduction 1
  1.1 BioMixer Overview 1
  1.2 Development With GWT 3
  1.3 Transition from Flash to SVG 3
  1.4 Scope of this Report 4
2.0 Rendering 5
  2.1 Creating SVG Elements with GWT 5
  2.2 Rendering the Graph 6
  2.3 Rendering Nodes 7
  2.4 Rendering Arcs 9
  2.5 Rendering Node Expanders 9
  2.6 Generalized Rendering 11
3.0 Interacting with the Graph 12
  3.1 Event Handling 12
  3.2 Node Event Handling 13
  3.3 Node Expansion 14
  3.4 Panning and Scrolling 14
4.0 Layouts 16
  4.1 Layout Architecture 16
  4.2 Circle Layout 16
  4.3 Tree Layouts 19
    4.3.1 Directed Acyclic Graphs 19
    4.3.2 Laying out the Directed Acyclic Graphs 20
    4.3.3 Vertical Tree Layout 21
    4.3.4 Horizontal Tree Layout 22
  4.4 Force Directed Layout 23
    4.4.1 Force Calculators 25
    4.4.2 Computing an Iteration 25
    4.4.3 Adjustments to Force Calculations 27
  4.5 Animating Layouts 27

4.6 Layout Execution Manager 28
5.0 Conclusions 29
6.0 Recommendations 30
References 31

List of Figures

Figure 1. BioMixer workspace example..1
Figure 2. A BioMixer visualization embedded in a third party website.................................2
Figure 3. Creating a green rectangle with SVG from Java code...5
Figure 4. Inspection of the graph with Firebug...7
Figure 5. A node before its expansion tab is clicked...8
Figure 6. A node with its expansion tab clicked..10
Figure 7. Nodes arranged by a circle layout...17
Figure 8. Calculation for determining the potential radius of the circle layout..................18
Figure 9. Nodes arranged by a vertical tree layout..22
Figure 10. Nodes arranged by a horizontal tree layout..23
Figure 11. Nodes arranged by a force directed layout..24

Summary

BioMixer is a web-based tool that enables visualization of biomedical ontologies. An ontology is basically a collection of concepts from a specific domain, as well as the relationships between these concepts. The primary form of visualization in BioMixer is a graph viewer, where concepts from ontologies are shown as nodes, and relationships between them as arcs. Mousing over nodes can provide additional information about concepts, such as definitions and synonyms.

At the start of this workterm (January 2012), the graph viewer was implemented using Adobe Flash. The possibility of re-developing it was being considered for several reasons. Firstly, Stanford University, who funds the project, wishes to promote the use of open standards. Secondly, Flash is not supported on Apple devices such as the iPad. Finally, the code for the Flash implementation is written in Actionscript and not directly part of the rest of the project. The integration process is fairly complex, and it provides an obstacle for developers of BioMixer (who predominantly code in Java) when they want to develop new features, fix bugs, etc.

For the reasons stated above, it was decided to re-develop the graph viewer using the open standard SVG. The re-implementation of the graph viewer also provided the opportunity to improve the flexibility and extendibility of the rendering and layouts. It will provide a more adaptable basis for further development of new visualizations.

Glossary

Actionscript – a programming language for Adobe Flash.

Adobe Flash – a multimedia platform which provides interactivity for web pages.

Canvas – an HTML element for creating two-dimensional raster graphics.

CSS – stands for Cascading Style Sheets. They provide a way to provide styling and modify the presentation of HTML pages.

Div – an HTML tag used to group other elements for styling, etc.

Firebug – an add-on for the Firefox web browser which provides web development tool support.

GWT – stands for Google Web Toolkit. It allows web applications to be developed using Java. The Java code is then cross-compiled to Javascript before deployment so that it can run in a web browser.

HTML – stands for HyperText Markup Language. It is the standard for representing web page contents on the Internet.

iframe – an “inline frame” in an HTML document. It allows another HTML document to be embedded within a page.

Java – an object-oriented computer programming language.

Javascript – a programming language primarily used for providing dynamic content in web pages.

Ontology – a collection of concepts from a specific domain, and the relationships between these concepts.

SVG – stands for Scalable Vector Graphics. It is a convention for specifying two-dimensional vector graphics using an XML based language.

XML - stands for extensible mark-up language. It is a standardized format for encoding documents in a form readable by computers, using text data. It can be used to represent all sorts of data structures.

1.0 Introduction

This section provides some background information about the BioMixer project and motivation for the development of an SVG graph viewer.

1.1 BioMixer Overview

BioMixer provides an environment for visualizing and exploring biomedical ontologies. It exists in two main forms. The first is the full environment, an example of which is shown in Figure 1.

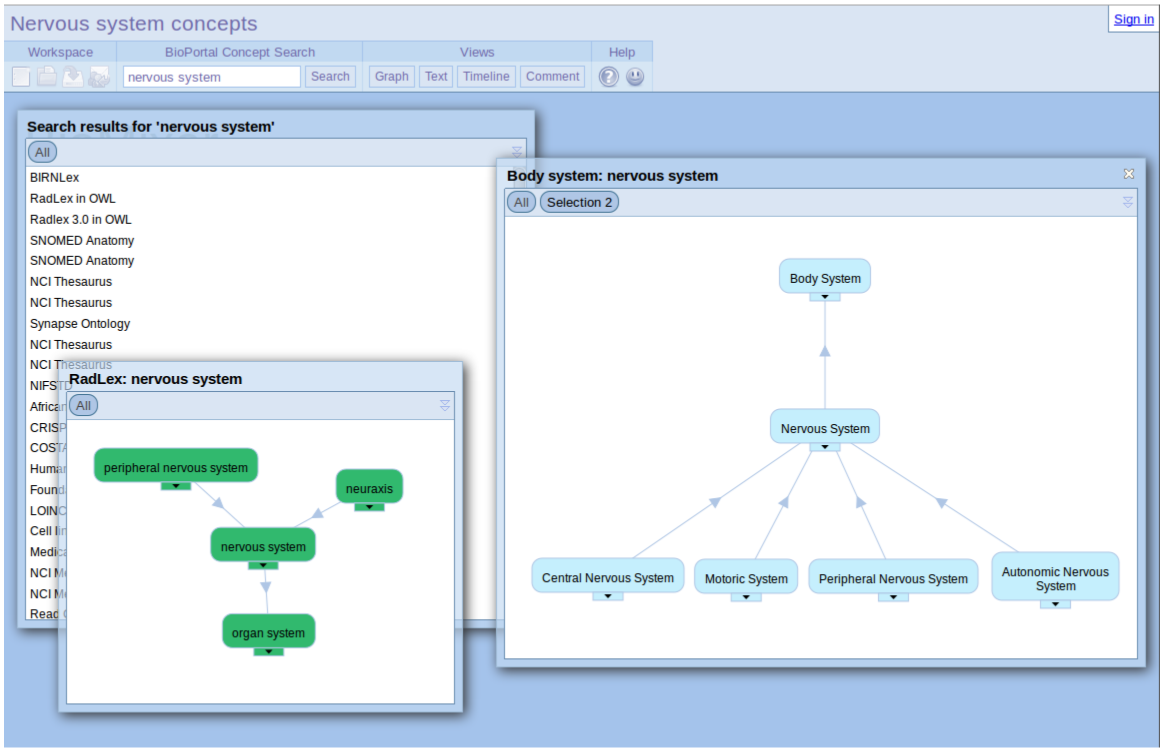
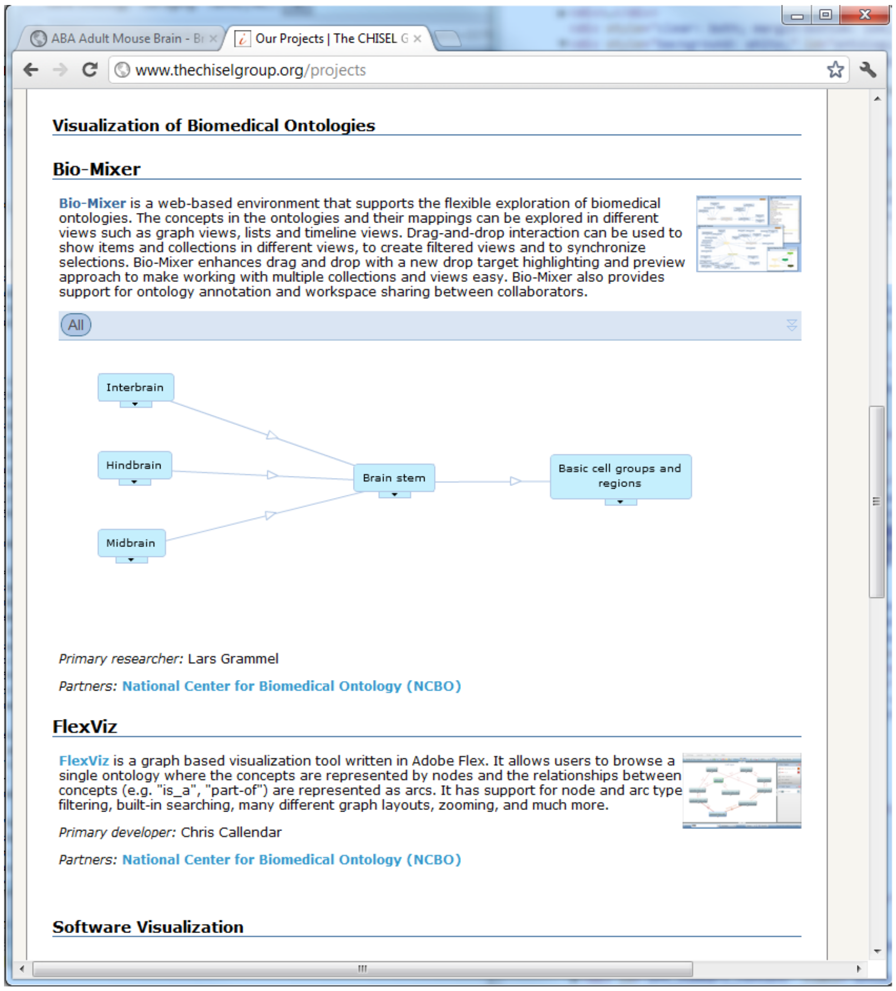


Figure 1. BioMixer workspace example.

The full environment supports searching for concepts and dragging and dropping the results into views to be visualized. The primary visualization type used is a graph where nodes represent concepts in an ontology, and arcs represent relationships between concepts. Multiple views can exist at once, and selected concepts can be dragged between views. These selections can be coordinated across views. For example, highlighting a concept in one view will highlight it in any other views in which it is present. This coordination of views can help the user examine different aspects of related data in different views that they manipulate independently. A user's workspace can be saved, reopened, and shared once the user has logged in.

The second form in which BioMixer can be used is as an "embedded" view. These visualizations consist of a single view encapsulated in an iframe which can be embedded into third party sites. An example is shown in Figure 2.


ABA Adult Mouse Brain - Br
Our Projects | The CHISEL G
www.thechiselgroup.org/projects
All
Interbrain
Hindbrain
Midbrain
Brain stem
Basic cell groups and regions

Visualization of Biomedical Ontologies

Bio-Mixer

Bio-Mixer is a web-based environment that supports the flexible exploration of biomedical ontologies. The concepts in the ontologies and their mappings can be explored in different views such as graph views, lists and timeline views. Drag-and-drop interaction can be used to show items and collections in different views, to create filtered views and to synchronize selections. Bio-Mixer enhances drag and drop with a new drop target highlighting and preview approach to make working with multiple collections and views easy. Bio-Mixer also provides support for ontology annotation and workspace sharing between collaborators.

Primary researcher: Lars Grammel

Partners: National Center for Biomedical Ontology (NCBO)

FlexViz

FlexViz is a graph based visualization tool written in Adobe Flex. It allows users to browse a single ontology where the concepts are represented by nodes and the relationships between concepts (e.g. "is_a", "part-of") are represented as arcs. It has support for node and arc type filtering, built-in searching, many different graph layouts, zooming, and much more.

Primary developer: Chris Callendar

Partners: National Center for Biomedical Ontology (NCBO)

Software Visualization

Figure 2. A BioMixer visualization embedded in a third party website.

1.2 Development With GWT

BioMixer has been developed using GWT (the Google Web Toolkit), which allows web applications to be developed using Java which then gets cross-compiled to Javascript so that it can be run in a web browser. This allows Java's tool support and testing frameworks to be utilized in development. However, only a subset of Java's runtime library is supported [1]. Therefore, there is a fundamental mismatch between the language being used and the web browser's capabilities, which developers must keep in mind.

1.3 Transition from Flash to SVG

Originally BioMixer used a graph viewer written in Actionscript. It was developed separately from BioMixer with different tools, which was an obstacle in further development and bug fixing. Instead of setting up an environment for developing in Actionscript and also learning the language, it was decided that it might be more beneficial to the project in the long run to develop a more closely integrated graph viewer. Such a transition would likely be required in the future anyway for compatibility reasons, so it didn't make sense to invest more time into software that would need to be replaced. Therefore the search for a new graphics technology was begun.

SVG is a convention for specifying two dimensional vector graphics. It is an open standard which has been developed by the World Wide Web Consortium [2]. Stanford University, which funds development of the tool, is interested in using open standards whenever possible. Open standards also allow the software to be run on a wider range of devices where Flash is not supported, such as Apple's products like the iPad. This could be even more important for maintaining compatibility in the future as closed standards may fall out of favour, but open standards should continue to be supported nonetheless.

In recent years SVG has been gaining in popularity. Modern browsers including Mozilla Firefox, Google Chrome, Safari, Opera and Internet Explorer 9 can render SVG directly. Earlier versions of Internet Explorer have no default support for rendering SVG, but there are some plugins available.

1.4 Scope of this Report

In addition to its function as a co-op work term report, this report is intended to provide some documentation of the graph viewer for other developers who will work on the system in the future. Therefore it may occasionally go into more detail in some subjects that may be useful for those developers. Nevertheless, it tries to remain a fairly high level overview.

2.0 Rendering

This section covers the rendering of nodes, arcs, and other graph elements. Here rendering means the process of making these elements visible to the user in the web browser.

2.1 Creating SVG Elements with GWT

Since the graph contents are dynamic as the user explores and expands concepts, static SVG files cannot be used. New SVG elements have to be created on the fly, and added to the web page without any reloading. This is the sort of scenario where Javascript is required.

Since the project is built using GWT, SVG abstractions were developed which allow the creation and manipulation of SVG elements from Java code. Internally, however, the abstractions are still using Javascript. This is achieved through GWT's ability to directly include Javascript code from within Java code (called Javascript native interface or JSNI) [3].

A short snippet of Java code demonstrating how the SVG abstractions are used is shown in Figure 3.

```
SvgElement baseContainer = svgElementFactory.createElement(Svg.RECT);
baseContainer.setAttribute(Svg.X, 0.0);
baseContainer.setAttribute(Svg.Y, 0.0);
baseContainer.setAttribute(Svg.WIDTH, 100.0);
baseContainer.setAttribute(Svg.HEIGHT, 50.0);
baseContainer.setAttribute(Svg.FILL, “green”);
```

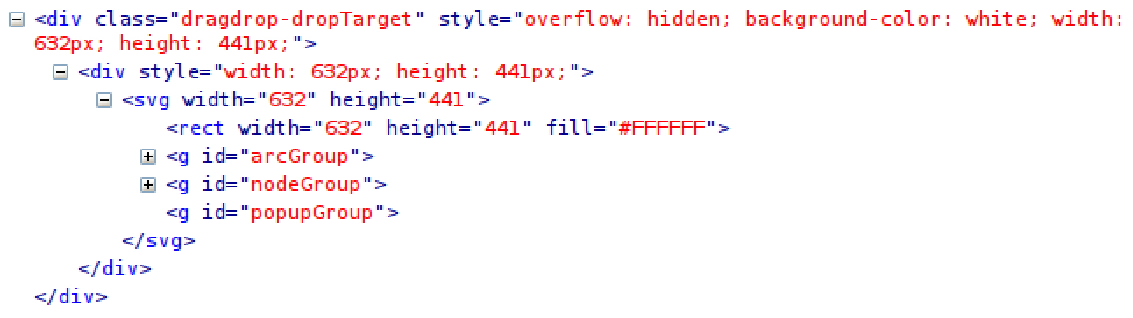
Figure 3. Creating a green rectangle with SVG from Java code

2.2 Rendering the Graph

There are currently three types of objects that can be drawn on a BioMixer graph: nodes, arcs and node expanders. Nodes represent concepts in an ontology, arcs are the relations between them, and the node expanders provide options for displaying other nodes related to an existing node. In addition, there is also a background element which provides a white background for the graph.

The SVG specification does not include a z-index for determining which element is in the foreground in case of overlaps. Instead SVG uses a “painters model” [4], where the order that SVG elements are drawn is determined by their order in the SVG document fragment. Elements are painted in the order they show up in the SVG document, so those coming later will paint over top of any previously drawn elements with overlapping locations.

The background of the graph should be the first thing in the SVG document so that it does not cover anything up. For simplicity, arcs are drawn between centres of nodes. However, the arcs should not be drawn on top of the nodes. Therefore the arcs must always be earlier in the SVG document than the nodes. Likewise, any popup options provided by node expanders should show up over top of anything else on the graph, so they should be after the nodes. To help enforce this ordering SVG group (g) elements are created for arcs, nodes, and popups in that order. Elements can then be added to these groups and they will still be drawn in the proper order. For example, even if an arc has been added after an overlapping node, the node will still be drawn on top because its grouping is further down the document. These grouping structures and their order are illustrated by Figure 4 which is a screenshot of Firebug's display of the SVG elements in XML format.



```
⊟ <div class="dragdrop-dropTarget" style="overflow: hidden; background-color: white; width:
  632px; height: 441px;">
    ⊟ <div style="width: 632px; height: 441px;">
        ⊟ <svg width="632" height="441">
              <rect width="632" height="441" fill="#FFFFFF">
            ⊞ <g id="arcGroup">
            ⊞ <g id="nodeGroup">
              <g id="popupGroup">
          </svg>
      </div>
  </div>
```

Figure 4. Inspection of the graph with Firebug.

2.3 Rendering Nodes

The previous graph viewer drew nodes as labeled rectangles, so that is how nodes have been implemented in the new graph viewer. A node graph element is shown in Figure 5. The label is the display name of the concept which the node represents. The rectangle should be sized appropriately according to this text. In order to do this, a way of measuring the text as it will display in the browser is required.

Several methods of measuring text size have been implemented. The fastest one creates a Canvas HTML element containing the text with a specified font and then uses the Canvas' metrics to find the size in pixels. This Canvas implementation can only measure the width, so it delegates measuring the height to another text size calculator. Another implementation creates an SVG element with text on it, has it rendered by the browser, then retrieves the boundary box of the element (a supported operation on rendered SVG elements).

Once the text size is known, the node is drawn with the surrounding rectangle having some padding space on each side. The text is centered within the rectangle. Since some labels can be quite long, there is support for wrapping the text in a node. A threshold width can be specified. The label is split on whitespace, and each word is measured. Additional words are added to the line until adding the next word would put the line's length past the threshold value. At that point a new line is begun. SVG supports stacked lines of text through tspan elements. While each line is less than the threshold, they do not necessarily have the same length. Therefore each line is centered in the surrounding rectangle.

The other main portion of the node is its tab at the bottom which brings up options for expanding further nodes. It is drawn with an additional rectangle centered at the bottom edge of the node. It has a small triangle pointing downwards which is drawn using an SVG polygon element.

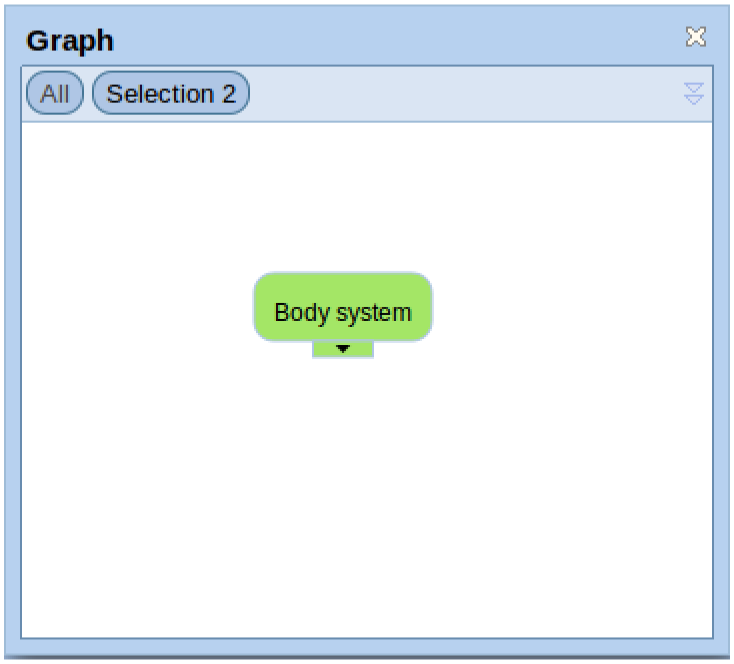


Figure 5. A node before its expansion tab is clicked.

Once the node's components have been created, they are stored in a Java class which provides methods for adjusting all sorts of attributes such as position, node background colour, node border colour, text colour, text font, etc.

2.4 Rendering Arcs

Arcs are currently rendered as straight lines using SVG's line elements. The start and end position are the centres of the source and target nodes. In order to make sure the arc positions stay up to date, a rendered node keeps track of arcs connected to it. Whenever the node moves, it tells its connected arcs to update to the new location.

If an arc is directed, it gets an arrowhead drawn using an SVG path element with a fill colour applied. This arrowhead gets rotated and translated as the arc updates in response to connected node movements.

A Java class stores the arc components and allows attributes such as the arc's colour and thickness to be adjusted. The arc can also be made dashed or solid using the SVG stroke-dasharray attribute which specifies the lengths of dashes and the gaps between them.

2.5 Rendering Node Expanders

The node expanders provide options for adding new nodes to the graph which are related in some way to the node being expanded. The handlers needed to perform these actions are registered with the graph viewer along with a short label. Currently the system has "Concepts" and "Mappings" expanders. Figure 5 above shows a screenshot of a node before having its expansion tab clicked, and Figure 6 shows it after having the expansion tab clicked.

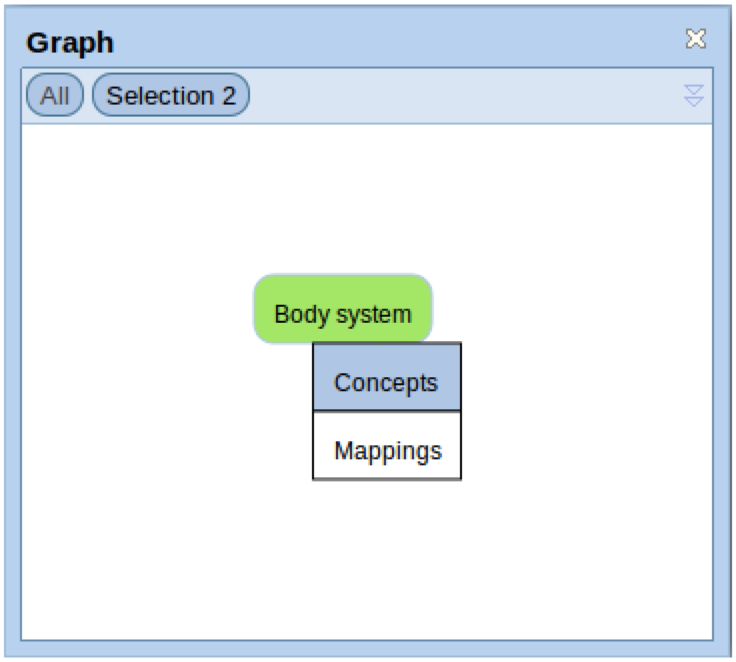


Figure 6. The node of Figure 5 with its expansion tab clicked and the first expansion option highlighted.

As can be seen in Figure 6, the node expander reuses the idea of boxed text from the nodes. The expansion options are displayed as stacked, boxed text objects which have had their width adjusted to be that of the widest option, so that they can have a common width.

2.6 Generalized Rendering

The new graph viewer was initially implemented to simply replicate the previous graph viewer's rendering style. After this was accomplished, the system was refactored to extract the rendering process. This allows other SVG renderings to be developed and swapped in. Most of the code is even decoupled from the SVG components, so that another rendering medium, such as Canvas, could be used and only the isolated portions which need to know about the rendering medium would have to change.

3.0 Interacting with the Graph

A key aspect of BioMixer's visualizations is their flexibility, which requires the ability to interact with the graph. There are many interactions with the graph supported, a subset of which will be discussed here.

3.1 Event Handling

Interactions between the user and the graph are managed by event handlers. These event handlers are registered on an object to listen for certain types of browser events. When they detect an event has occurred, they trigger some sort of response. Some common browser events are:

1. Mouse down: the mouse button has been pushed down but not yet released.
2. Mouse up: the mouse button has been released after being pushed down.
3. Mouse click: the mouse button has been pushed down and released. This event gets fired at around the same time as a mouse up event.
4. Mouse move: the mouse has been moved on the screen.
5. Mouse over: the mouse has been brought hovering over top of an item.
6. Mouse out: the mouse was hovering over an item, but has moved off it.

Each SVG element can have one event handler registered on it. This event handler will listen for one or more of the above events, and take some action in response.

3.2 Node Event Handling

There are many interactions that need to be supported with nodes. These include showing additional details about a node when mousing over it, bolding the node when adding it to a selection through a mouse click, and moving the node when it is dragged (mouse down plus mouse move).

External event handlers can be supplied for some of these actions. For example, the handling of a mouse over action is specified by a handler registered in an event bus in the graph display controller. When the event handler on the node being moused over detects the browser event, it creates a "node moused over" event which it fires into the event bus. The external event handler receives this and some code is run in response. In this way, the action taken in response to an event is decoupled from the actual occurrence and detection of the event.

Events such as node movement are handled locally in the graph display. Whenever an event related to node movement occurs, it is forwarded to the node interaction manager. The node interaction manager maintains some state about whether a node is currently "mouse down", as well as the last known mouse coordinates. Then when a mouse move event comes in, it can compare the new coordinates with the previous coordinates to produce a position delta. The node can then be updated to reflect this position delta. If no node is "mouse down" then no node has its position modified by the mouse move event, but the last known mouse position will still be updated.

This node interaction manager does more than receive events forwarded by nodes. If that's all it did, then if the user moved their mouse fast enough that it went off the node, further mouse movements would not update the node's position. To remedy this problem, there is also a “view wide” event listener. It is registered on the root SVG element, and will notify the node interaction manager of mouse move events even if they are not over top of a node.

3.3 Node Expansion

Several interactions are required to expand a node. First its expansion tab must be clicked. In response to this, the node expander gets rendered. The node expander provides several options, each of which has event handling associated with it. This event handling allows options to be highlighted when they are moused over, and detects when one is selected through a click event.

When a node expansion option is clicked, the corresponding node expansion handler is notified. The node expansion options are only rendered for node expansion handlers that were registered when the graph was created. These expansion handlers hide the details of retrieving the related nodes and adding them to the graph.

3.4 Panning and Scrolling

Sometimes it is useful to be able to move all elements on the graph at once. This could be used to manually centre the graph contents, for example. Panning is detected by registering an event handler on the background element. Similar to detecting dragging of a node, it detects mouse down plus mouse move events on the background. In response, all graph elements are shifted by the vertical and horizontal deltas calculated for the user’s drag.

While panning it is possible to push nodes off the visible graph area. These nodes should not get lost, they should be accessible via scrolling. Scrolling is supported by using CSS's support for scrollbars on the SVG element's container divs.

4.0 Layouts

Once nodes and arcs have been rendered on a graph, it is important to display them in a way which promotes understanding and analysis. To do this, different layout algorithms were developed. Different layouts may emphasize different aspects of the data.

4.1 Layout Architecture

The layout graph, including layout nodes and layout arcs, is not tied to a specific rendering. They have their own interfaces which allow the layouts to be developed and tested independently.

There are two main types of layouts that are supported by the system: continuous and non-continuous. A non-continuous layout runs one computation based on the current state of the graph and then finishes. A continuous layout on the other hand computes many iterations as part of the same computation. In each iteration it performs the next step of the calculation based on the state of the graph that resulted from the previous iteration.

4.2 Circle Layout

The circle layout organizes nodes in a circular manner, with the arcs crossing through the circle. Figure 7 shows a typical example.

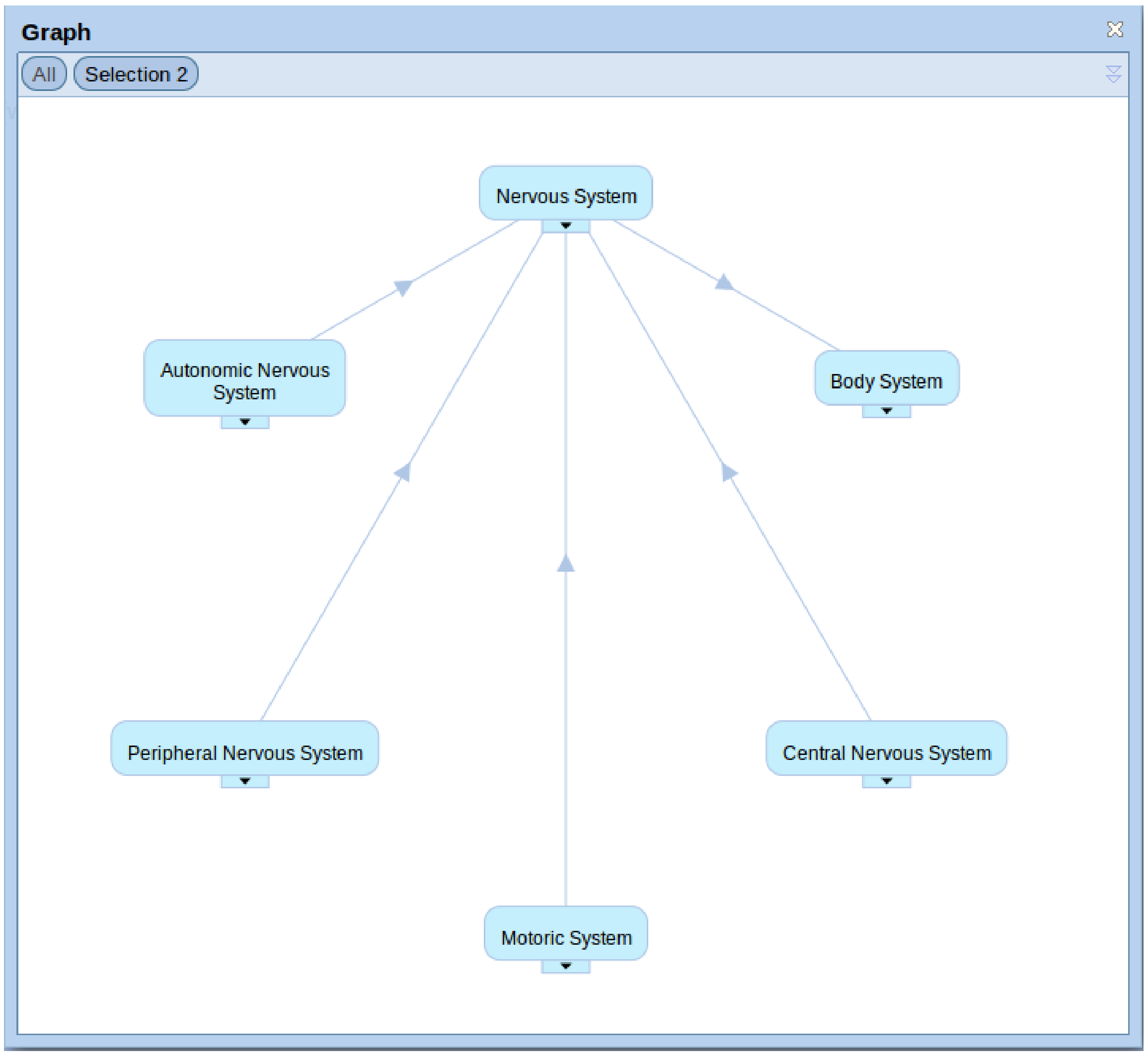


Figure 7. Nodes arranged by a circle layout.

This is a non-continuous layout. When run, the layout can immediately calculate the final positions of all the nodes. It does this calculation by checking how many nodes there are, and using this to determine what angular separation should be between nodes with respect to the centre of the circle.

An angle range may be set for this layout. For example, if the angle range was (90, 270) then the nodes would be laid out in a half-circle with the concave face pointing upwards. In Figure 7 the angle range has been left at its default (0, 360). Therefore, the angular separation between each of the 6 nodes (with respect to the centre of the circle) should be 360°/6 = 60°.

Once the angular separation between nodes has been determined, they need to be positioned on a common radius at these angles. The radius is determined by taking the minimum of the potential radius for the x axis and the potential radius for the y axis. The potential radius for an axis is determined by dividing the available graph length in that dimension by two, then subtracting off a padding amount (say, 5% of the total graph length in that direction). There also needs to be a compensation for the varying sizes of nodes so that a large node does not go off the edge. This is done by further subtracting off half of the width/height of the widest/tallest node, depending on the dimension for which the calculation is being performed. Figure 8 shows a short Java snippet with this calculation for determining the potential x radius. A similar calculation is performed for the y axis, then the minimum is used as the radius.

```java
double radiusX = graphWidth / 2 - horizontalPaddingPercent * graphWidth
            - LayoutUtils.getMaxNodeWidth(allNodes) / 2;
```

Figure 8. Calculation for determining the potential radius of the circle layout.

After the radius and angular separation between nodes has been calculated, the layout iterates through the nodes. For each node the angle gets incremented by the angular separation, and it then gets placed at the radius length from the centre of the graph using simple trigonometry to find the actual x and y coordinates.

4.3 Tree Layouts

Like the circle layout, the tree layouts are non-continuous. Two tree layouts have been developed, the vertical tree layout and the horizontal tree layout. They are basically the same layout with different orientations, so they share most of their code. Each of them can also have its orientation flipped along its primary axis; that is, the vertical tree layout can be set to point upwards or downwards, and the horizontal tree layout can be set to point left or right.

4.3.1 Directed Acyclic Graphs

The tree layouts work best on directed acyclic graphs. A directed acyclic graph contains only directed arcs and no cycles. This means that it is not possible to start at some node and follow some path to eventually get back to that node. Directed arcs which indicate a hierarchy point from the child to the parent ("is a" relationship). Therefore a node which has only children (only arcs pointing towards it, none away from it) is considered a "root".

The first step in the tree layouts is to find all directed acyclic graph structures on the graph. Initially, each node is considered to be a potential root. Then all the arcs are iterated over. From each arc we can retrieve the source and target nodes. The source node is added as a child of the target node, and removed from the list of potential roots. Once all the arcs have been processed, we have one or more directed acyclic graph structures.

It is possible that two roots could share some common descendants, in which case we want to merge the two graph structures. In order to do this we take the intersection of two root’s descendants. If the intersection contains any nodes, the graph structures get merged. The result is a collection of directed acyclic graphs which may have multiple roots.

4.3.2 Laying out the Directed Acyclic Graphs

Once the directed acyclic graph (DAG) structures have been identified, the process of positioning them on the screen can begin. In this discussion, reference will be made to the primary and secondary dimension of the graph. For the vertical tree layout, the y direction is the primary dimension and the x direction is the secondary dimension. For the horizontal tree layout it is the other way around.

The first step is to portion the screen into chunks for each of the DAGs. Each DAG will be given the full available space along the primary dimension. Currently each DAG is then allotted an equal portion of the secondary dimension.

The DAG structures can be thought of as having distinct layers. For example, the root would be on the top layer, below it would be a layer with its children, then the children’s children, and so on. Methods can then be written to retrieve nodes by level. If a node can be reached along multiple paths of different lengths, it is put into the furthest possible level that it qualifies for (based on the longest path from it to the root). This notion of layers in the DAGs forms the basis for the layout structure.

Since the space available for each DAG is known, and the DAG can be accessed layer by layer, an appropriate spacing between nodes can now be determined.
In the primary dimension:

spacing = available primary dimension / (# of levels in the DAG + 1)

In the secondary dimension:

spacing = available secondary dimension / (# nodes in most occupied level + 1)

Once these spacing values are known, the DAG can be processed level by level, incrementing the secondary dimension between nodes of the same level, and incrementing the primary direction when moving to a new level. The next sections give some description of this in more concrete terms for the two implementations.

4.3.3 Vertical Tree Layout

For the vertical tree layout, the primary dimension is the y axis and the secondary dimension is the x axis. Therefore the levels of the DAG are stacked on top of each other vertically, with nodes of the same level being horizontally aligned. An example with a single DAG is shown in Figure 9.

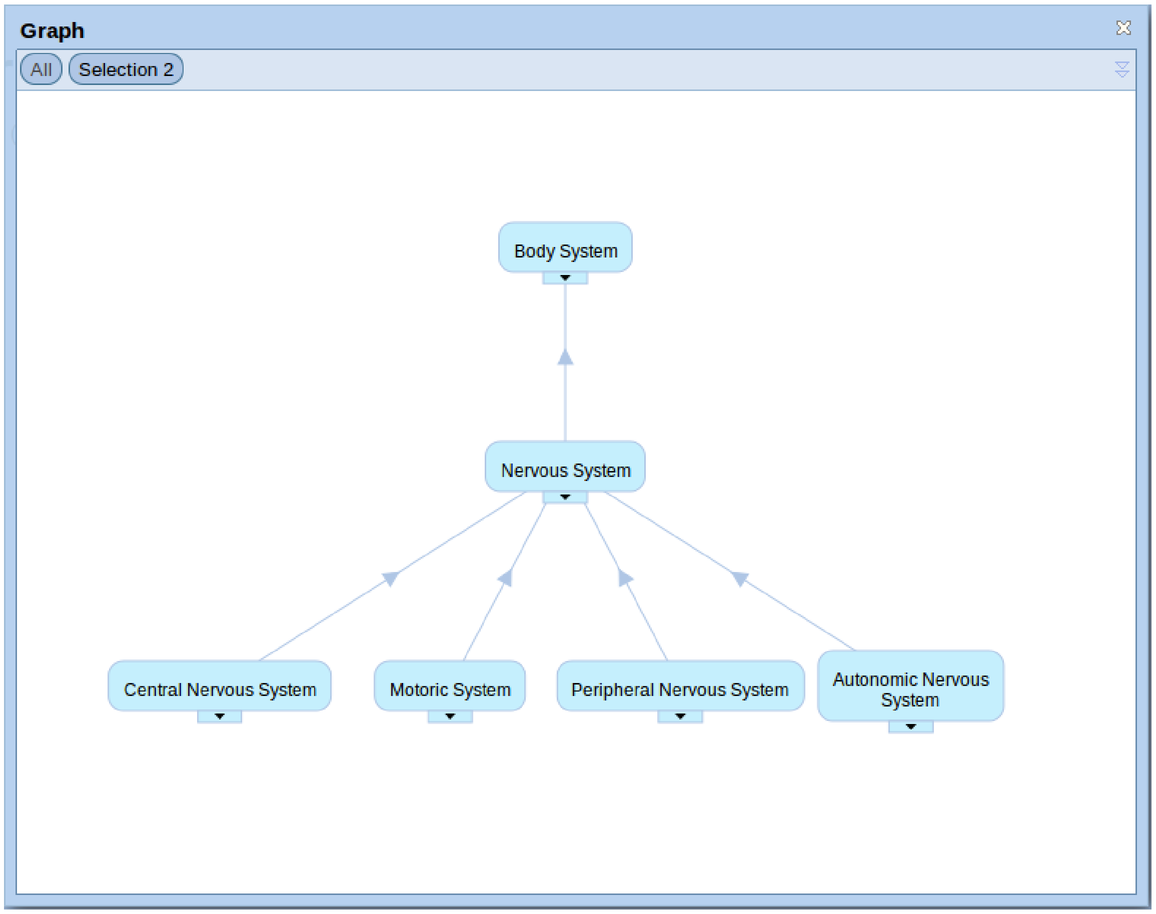


Figure 9. Nodes arranged by a vertical tree layout.

The graph could also have been laid out with the arcs pointing downwards by passing in a Boolean flag to reverse the end of the graph used as the starting point.

4.3.4 Horizontal Tree Layout

The horizontal tree layout uses the x axis as its primary dimension, and y axis as the secondary dimension. In this case nodes in the same level of the DAG are stacked on top of each other. Figure 10 shows an example with a single DAG.

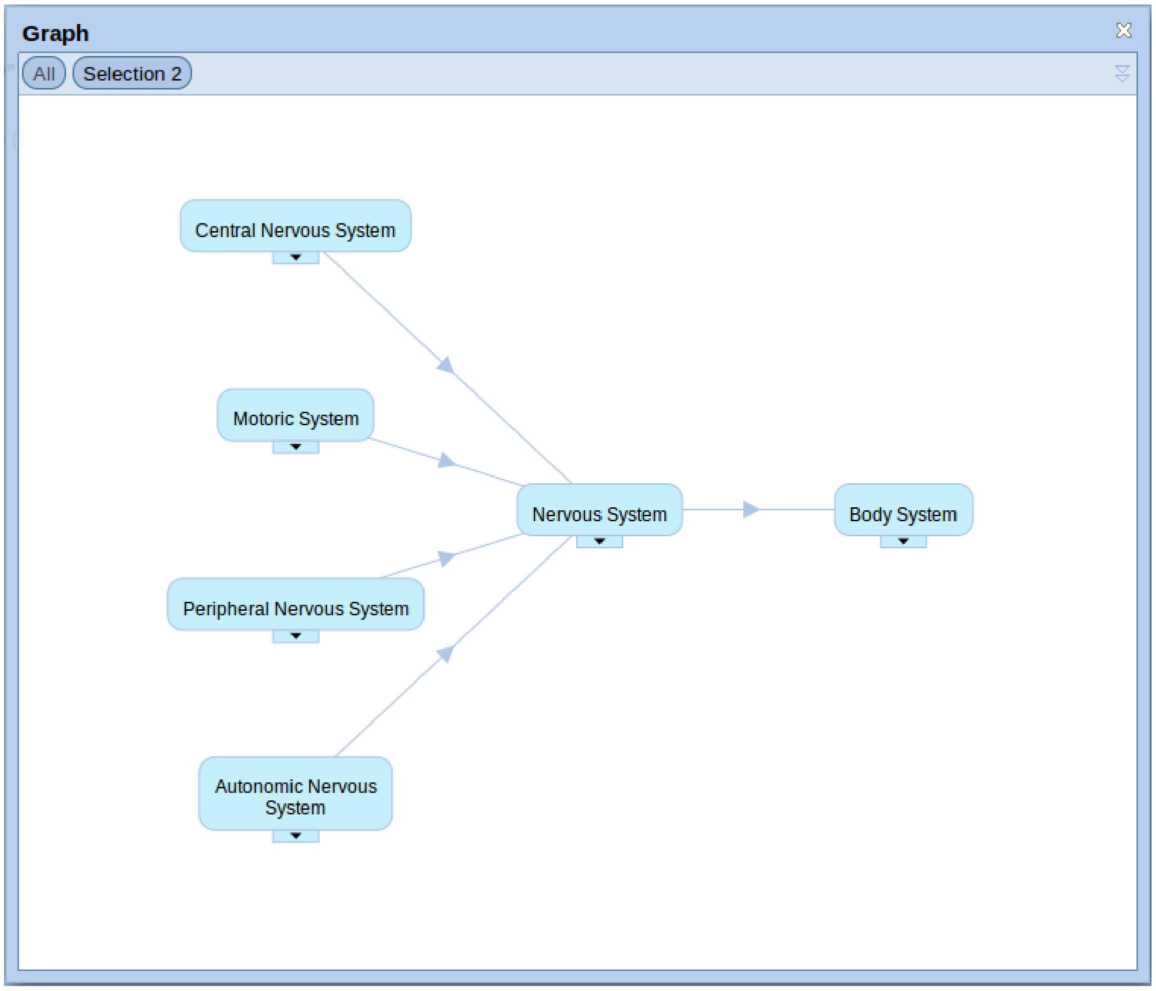


Figure 10. Nodes arranged by a horizontal tree layout.

Again, the direction of this layout could have been flipped to have the arrows pointing left by having the nodes placed starting from the other direction.

4.4 Force Directed Layout

The force directed layout is different from the other layouts in that it is continuous; that is, it computes multiple iterations in order to determine the final locations of the nodes. The basic idea of the force directed layout is to position the nodes so that edges have fairly equal length and so that there are minimal edge crossings. This is done by calculating forces between nodes which may have a net attraction or repulsion effect. These forces then cause movement of the nodes. This process of calculating forces then updating position is repeated until the graph approaches a state of equilibrium. Figure 11 shows the end result of running a force directed layout on several nodes.

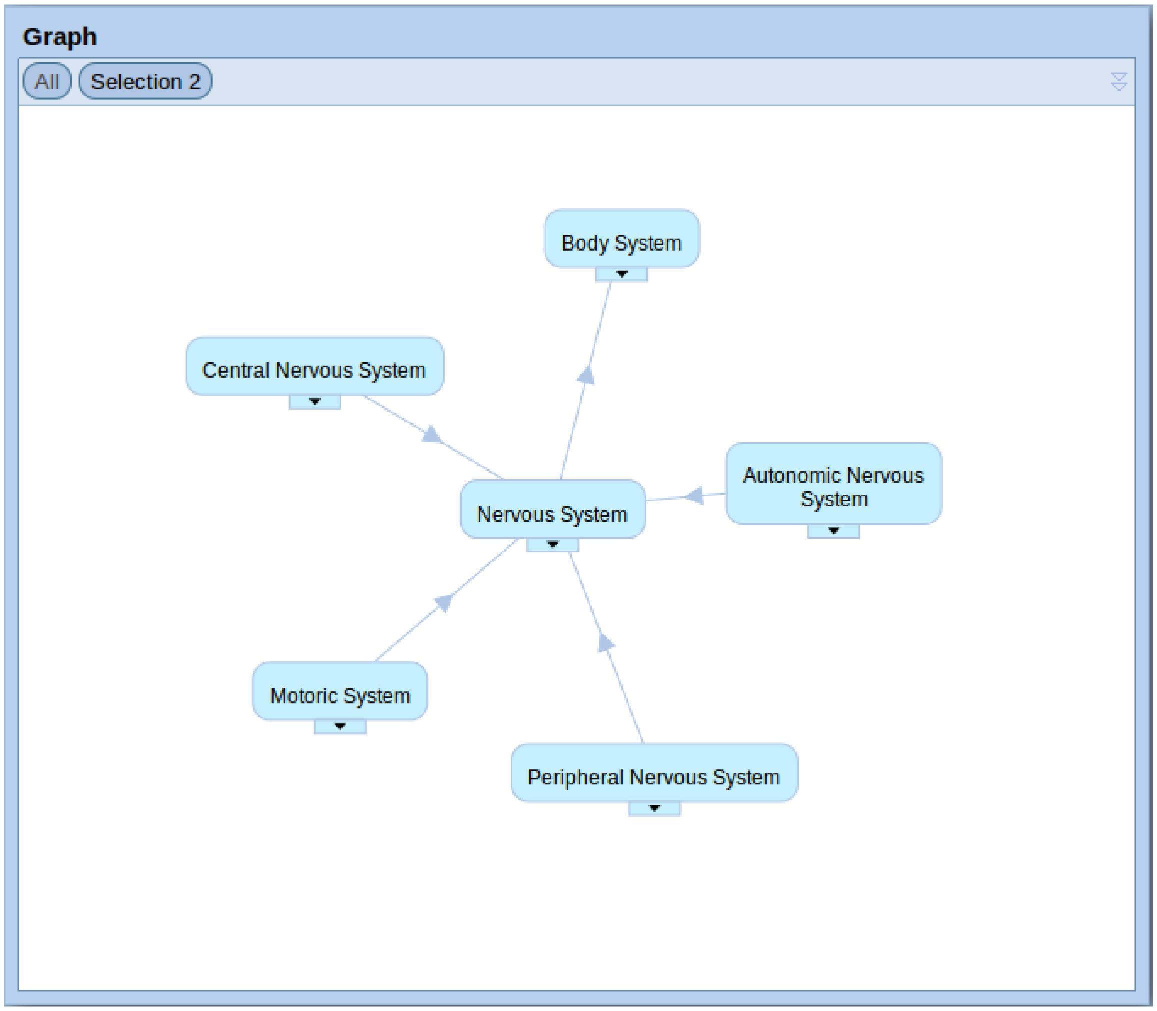


Figure 11. Nodes arranged by a force directed layout.

4.4.1 Force Calculators

There are many ways in which the forces between nodes could be calculated. The calculations necessary to determine the force between two nodes are encapsulated in force calculators. The rest of the force directed layout algorithm is not tied to a specific implementation, it can use any force calculator supplied. Typically the force calculator supplied should be a composite of an attractive force calculator and a repulsive force calculator. This composite will sum the attractive and repulsive forces between two nodes.

The force calculators currently being used are based on Fruchterman and Reingold's [5] equations. These equations are:

attraction force = $f_a(d) = d^2 / k$,

repulsion force = $f_r(d) = -k^2 / d$

where d is the distance between the two nodes and k is the optimal edge length, given by:

k = C*sqrt(graph area / number of vertices),

where C is some constant.

There is a repulsion force between all nodes, but only an attraction force if they are connected by an arc.

4.4.2 Computing an Iteration

In each iteration of the force directed layout computation, the net force on each node is calculated. This net force will be the vector summation of all the attractive and repulsive forces from all the other nodes on the graph.

Each node also has a dampening factor associated with it. It is possible for different nodes to have different dampening factors. The dampening factor is a number between 0 and 1, which starts out at 1 and then is multiplied by a dampening constant each iteration. The net force on a node is scaled down by its dampening factor to get its displacement vector. Increasing this dampening factor for all nodes each iteration gradually decreases the distance that nodes are able to move. It helps prevent nodes which have reached a good configuration from moving back towards one that is not as good. This process in general is called simulated annealing, and is often used when trying to find a good approximation for a global optimum when it might be too expensive to do an exhaustive enumeration to find the true global optimum. Fruchterman and Reingold have a similar mechanism in their algorithm which they call temperature.

In addition to dampening, a node's displacement vector may be further restricted if it would be leaving the graph bounds. If a node would be pushed off the graph, its movement is restricted to the position at which it would be crossing over the graph's boundary.

The algorithm continues computing iterations until the average node displacement (after dampening and boundary restrictions) is below a threshold value. The average displacement is used for this threshold since if total displacement was used, a graph with many nodes moving just a tiny amount would stay above the threshold longer than a graph with just a few nodes each moving a larger amount.

4.4.3 Adjustments to Force Calculations

It was found that graphs with highly interconnected nodes could display excessive clumping due to the large number of attraction forces from all the arcs. To counteract this, the force calculators were changed to use a slightly modified version of Fruchterman and Reingold's equations. Instead of using some value such as 0.5 for the constant in
k = C*sqrt(graph area / number of vertices),
C is instead calculated to be the number of arcs on the graph divided by the number of nodes. Looking back at the attraction and repulsion force equations, this can be seen to have the effect of reducing attraction forces when the number of arcs is much greater than the number of nodes, and thereby preventing nodes from getting clumped together.

The other effect of using this scaling factor is that if there are two nodes with no arcs between them on the graph, there will be no force between them. Previously there would be no attraction force, so the repulsion force would tend to push them to opposite corners of the graph. It would still be desirable for them to exert some force on each other however, so if there are no arcs on the graph, the scaling factor is not left at zero, it is set to be some small value which allows the nodes to gently repulse each other.

<u>4.5 Animating Layouts</u>

In order to make the layout process look smooth, the movement of nodes to their newly calculated positions is animated. When setting up an animation, the start position, end position and desired animation duration must be known. The animation is then achieved by periodically checking what percentage of the animation's duration has elapsed. This percentage is used to update the node’s position to some intermediate value by using linear interpolation between the known start and end points.

4.6 Layout Execution Manager

An important consideration related to layouts is when to run them. The graph has been set up to fire an event whenever a node or arc is added to it. Other components may register themselves to receive these notifications. Currently there is a layout execution manager set up to receive notifications in this way. This allows it to update the layout whenever the graph contents change. This execution manager also allows new layout algorithms to be switched in as the default.

In section 4.4.2 it was mentioned that each node on the graph has its dampening coefficient maintained separately. The reason for this is that nodes can be added to the graph at different times. If a collection of nodes has already been laid out by the force directed layout, it is distracting to the user if they all move to completely different locations upon the addition of some new nodes. By maintaining separate dampening coefficients, nodes that have already been laid out will experience a much higher dampening effect than the nodes freshly added to the graph. This means the old nodes may move slightly, but mostly it will be the new ones moving to position themselves around the existing nodes.

5.0 Conclusions

The Flash-based graph viewer has been successfully re-implemented with SVG, an open standard. This implementation should ease further development and maintenance since it is written in the same language as the rest of the project. It has also been designed to allow different renderers to be developed and switched in without modifying the rest of the code. This means completely new node types and shapes could be developed for use in the visualizations. The differences could also be much deeper though. For example, the graph viewer is not dependant on SVG. One could provide rendering implementations that worked with some other technology, such as Canvas, if this became desirable, and the rest of the layouts, event handling and behaviours would stay the same.

6.0 Recommendations

The graph viewer is now ready to be tried out with different renderers. The BioMixer team has recently prototyped a visualization using the Javascript library d3 [6] which has entire ontologies (rather than individual concepts) represented as nodes. These nodes are circular, and their area is dependent on the number of terms they contain. With the graph viewer now ready, and the prototype approved, integration of this visualization into the BioMixer tool can be achieved by creating new renderers.

In addition, there is an issue with the tree layouts which warrants some further consideration. This issue is how to handle undirected edges. Undirected edges occur when a node's mappings are expanded. Since the tree layouts are currently based on the notion of there being directed acyclic graph (DAG) structures on the graph, it is not clear how undirected edges should be handled. Currently undirected edges are being ignored, which has the effect of placing the mapped nodes in DAGs by themselves.

One potential solution to the problem of undirected edges would be to lay out different nodes with different algorithms. Nodes in the DAGs would still be laid out with the tree layout, but any nodes connected by undirected arcs could be done using the force directed layout. This would keep the tree structure, and the nodes connected by undirected arcs would be clustered around the nodes in the tree to which they are connected. Ideally several options should be considered before deciding on the most desirable and practical course of action.

References

[1] Google Developers. “GWT JRE Emulation Reference.” [Cited 2012 April 29], available at https://developers.google.com/web-toolkit/doc/latest/RefJreEmulation.

[2] W3C Recommendation. “Scalable Vector Graphics (SVG) 1.1 (Second Editon).” [Cited 2012 April 30], available at http://www.w3.org/TR/SVG/.

[3] Google Developers. “GWT Coding Basics – JavaScript Native Interface (JSNI).” [Cited 2012 April 29], available at https://developers.google.com/web-toolkit/doc/latest/DevGuideCodingBasicsJSNI.

[4] W3C Recommendation. “SVG Rendering Model.” [Cited 2012 April 29], available at http://www.w3.org/TR/SVG/render.html#PaintersModel

[5] T. Fruchterman and E. Reingold. “Graph drawing by force-directed placement.” *Softw. – Pract. Exp.*, 21(11):1129–1164, 1991.

[6] Data-Driven Documents (D3). Homepage. [Cited 2012 April 28], available at http://d3js.org/