Mihir Samdarshi

February 1, 2019

BIOL 588 – Advanced Systems Biology Research

**Refactor of code to the Model-View-Controller paradigm improves** **GRNsight: a web application for visualizing small- to medium-scale gene regulatory networks**

Results

*Codebase*

Since its inception, GRNsight’s server and front-end code for the most part has been written in JavaScript. GRNsight uses Node.js as a server-side runtime environment, which delivers key information to the front-end code that runs the network visualization. Furthermore, since 2018, GRNsight has used Webpack in order to attain faster delivery of the service, as well as increased modularity. Webpack allows developers to bundle assets and create bundles within the code itself, thereby reducing the amount of front-end code that a browser must process in order to deliver an intended service (Koppers et al., 2015). The introduction of Webpack to bundle assets necessitated a move of the codebase from the fifth ECMAScript specification of JavaScript, which was released in 2009, to the sixth ECMAScript specification released in 2015, commonly known as ES6 or ECMAScript 2015. This allowed for the use of advanced features such as concise function declarations, as well as exports and imports. Furthermore, the addition of classes allowed for the introduction of possible further modularity. However, this also necessitated the addition of Babel, a compiler, to convert ECMAScript 2015 and newer specification code into a backwards compatible version of JavaScript in current and older browsers or environments. This was added to ensure the ECMAScript 6 code in the GRNsight codebase was backwards compatible in order to ensure user compatibility across the entire spectrum of our potential users. Additionally, the addition of Babel prepares the GRNsight codebase for the port of the codebase to the popular ReactJS framework, as it compiles JSX syntax, a portmanteau of HTML and JavaScript used heavily in ReactJS.

*Previous Architecture*

GRNsight was originally written in the fall of 2014, at which time the codebase was less than a third of the size than it currently is. Furthermore, considering that GRNsight at the time was limited to simply importing and displaying a Microsoft Excel sheet with weighted edges, very few files contained GRNsight’s codebase. At the end of 2014, the folder that contained GRNsight’s public-facing JavaScript code contained only five files. Only two of those files contained code written by GRNsight’s original developers. The other three files were necessary import, such as the jQuery library and a package to detect when all images have been loaded. As additions to front-end features accelerated, a similar structure was retained to the initial architecture of GRNsight, with each new feature getting a new file unless absolutely unnecessary. Immediately before the beginning of the current refactor work that started in May of 2018, nine files existed in the folder containing the web client code, of which two files were for external library imports.

In the previous architecture, all three aspects of the application were contained in singular files. These three aspects are what the user sees, the controls for how the user can modulate the application, and what the state of a feature is. We will illustrate this pattern using the graph physics that modulate GRNsight. Upon uploading an appropriate file, the user sees the graph with default physics enabled and, on the sidebar, a few sliders that display link distance and charge current values (among other modulation options); one of the controls for modulating the application is adjusting the charge on the nodes and the length of the edges connecting nodes; the state of the application consists of the current value of the link distance and charge sliders. Under the previous architecture, all three of these co-existed in the same file, and often times all three were contained in the same function. This architecture of GRNsight had the potential to be very problematic. The potential for conflicts between different aspects and features a green site was persistent. These clashes could lead to a detrimental user experience especially when modulating a certain combination of actions to the graph. The number of bugs that arose legend the decision to convert the entire front-end codebase to an architecture following the commonly used Model-View-Controller paradigm. This was done especially in anticipation of a move to ReactJS.

*Current Architecture*

GRNsight’s application of the Model-View-Controller paradigm (Figure 1) represents a similar web application interface design proposed by Lee and Rayfield (2001) which “displays information to the user… together with the Controller which processes the user's interaction… [and] Model… portion of the application that contains both the information represented by the View.” While following the basic architecture, GRNsight actually also separates the state of the application interface and the logic that controls changes to that application state.

The majority of GRNsight’s application interface code now resides in six files: setup-handlers.js, grnstate.js, update-app.js, setup-load-and-import-handlers.js, constants.js, and grnsight.js. As all of the files are designed so as to be modules, the grnsight.js file’s function serves to import and compile the various modules necessary for proper web application function into one complete file, a minified version of which is then served to a browser. The file constants.js acts as a module for the Webpack software to parse and populate multiple repeated variables across the majority of these files. This helps to improve code comprehensibility and allow for descriptive naming of these variables across the main components of the new architecture. The remaining four files contain the bulk of the application’s functional codebase.

The central store for the application state resides in an object called GRNstate, which is located in the eponymous grnstate.js. The file contains one central object with 30 properties, such as the default link distance and charge slider values, and two methods for getting and setting the network parsed by the server-side code. The state of every property that can be modulated, except for the size of the container of the graph, is stored inside this central model. It additionally contains a helper method to annotate the value of each of the edges in the network. This file can be best seen as the Model component. Update-app.js, serves best as the View component of the new GRNsight architecture. The update-app is triggered by changes detected by the Controller component, named in GRNsight as setup-handlers. Along with displaying the user interface of the GRNsight, update-app.js contains code to handle any potential actions allowed to be taken by the user. Given the various options for modulation, the code has been moved in such a way so as to allow for least conflict. No functions for the handling of different actions interact with each other, and none of the helper functions are called by functions handling different states of the application. The last central file to the new GRNsight architecture is named setup-handlers.js. This file contains a function named setup-handlers which receives changes from user and updates central store of app state. A secondary file imported into setup-handlers.js is named setup-load-and-import-handlers.js and contains the code necessary for the loading of the application container and the import of adjacency matrices into GRNsight. Setup-handlers.js contains a separate handler function for every action available to the user; each function changes a property of the GRNstate object and triggers the update-app function in order to execute the appropriate action. These two files would constitute the Controller portion of the MVC architecture. The remaining files contain the basic code necessary to create a graph with physics.

The order of movement was especially important in transitioning the entire codebase. Each distinct feature was relocated to the new set of files separately to ensure the least disruption in feature set. Additionally, each feature was tested completely to ensure that it fully worked before the work to transition the next feature to the new architecture. The first feature to be fully moved was displaying grey edges as dashed lines. This feature was moved first because it consisted simply of a checkbox that changed a simple appearance of graph edges. After accomplishing this, the order of the move followed a relatively simple logical progression, moving progressively larger and larger features. After establishing a process to follow when moving features, the set of functions that allowed users to modulate values that edge widths were normalized against were moved into update-app.js, grnstate.js, and setup-handlers.js. Next, the slider that controls the edge weight threshold to display edges as grey instead of colored. The next relatively easy feature to move over was the three radio buttons which allow the user to decide when to display edge weights. However, after these features were moved, the largest portion of the work followed, with the movement of the sliders that control the graph physics, as well as the recently added button which allows a user to automatically layout nodes in a grid. These features were heavily incorporated into the code which controls the main visualization of the graph. Thus, movement of these features often broke other features, requiring their move to be completed together. Finally, the last feature that a user is allowed to modulate that was moved was the node coloring feature, which existed solely in one file, making its movement easy, but arduous.

*ReactJS*

In the near future, a move of the front-end architecture to ReactJS is planned. ReactJS is a JavaScript library developed by Facebook designed especially for single-page applications (Vipul & Sonpakti, 2016, p. 2). The advantages of using ReactJS include increased modularity, and especially the creation of a “virtual” Document Object Model (DOM), or tree structure that every web page is encoded as. React creates an in-memory cache of the DOM, and, upon user-changes, only renders the sub-components that have been changed, allowing a developer to code the project with extreme modularity. However, a potential problem arises with a potential move to ReactJS, considering both d3.js, the visualization library that GRNsight uses, and the engine that ReactJS runs on depend on control of the DOM. A method must be researched and implemented to allow for simultaneous control of the DOM by both libraries. However, upon a cursory review of similar simultaneous implementations, it seems that packages exist which help to minimize conflict between the two. ReactJS is an important step for GRNsight to take as it allows developers to futureproof for the addition of new features.

*Visualization Improvements*

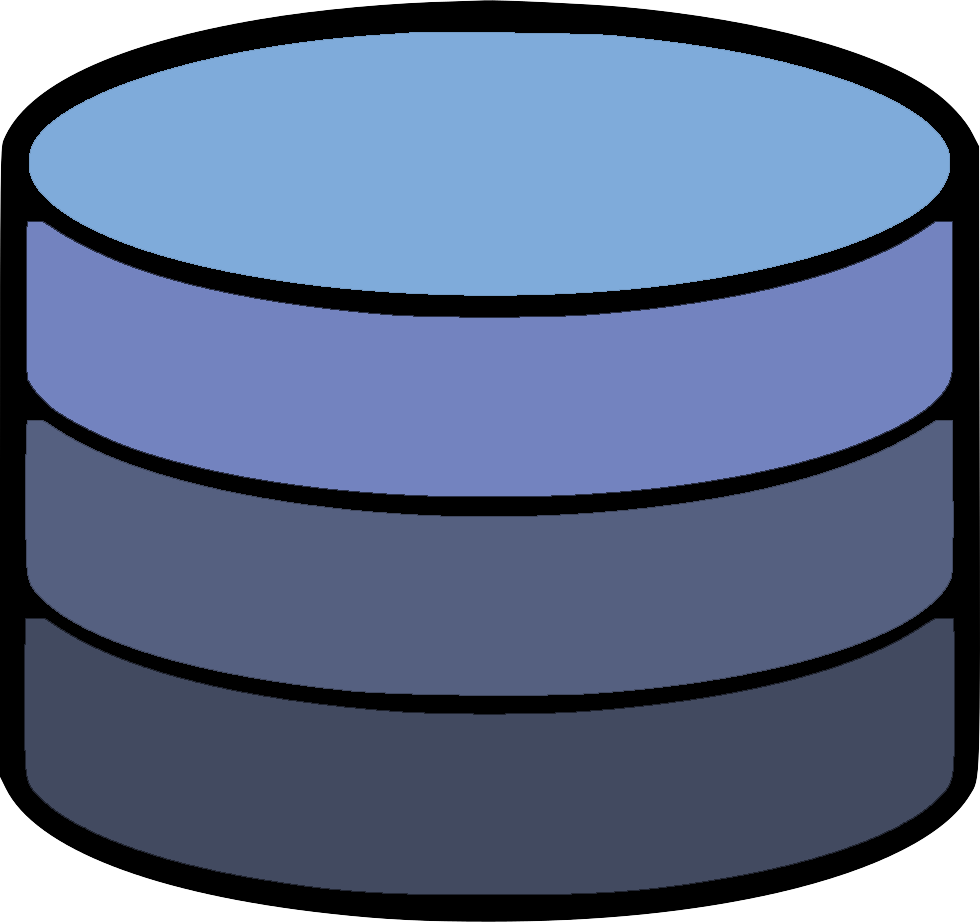
A couple of visualization features were added over the course of my work on the web application. The first was the ability of a user to control the edge weight threshold that it took to display edges as grey. Previously, edges were displayed as grey if their weight values were under a standardized five percent of the maximum edge weight value. Implementation of this feature was simple; a user-input slider was coded that displayed a percentage. If normalized values were under the corresponding percentage of the maximum edge weight value, the edges were displayed as a grey edge threshold (Figure 2). This was important to be able to focus only on edges with certain distinguished weights. Furthermore, a checkbox was implemented for color blind users to better distinguish a grey edge from a colored edge. This was done by adding a boolean variable that was updated as true upon change of the checkbox’s status. If the edge weight was under the threshold and the boolean held true, then the edge took on a style that appeared as if it was dashed (Figure 3). Finally, another feature which I worked on involved creating a feature that allowed users to normalize edge weight visualization against a user-inputted value, instead of the default largest edge weight value. A text box was created with a set and reset button. The reset button was set to default to the largest edge weight value. Inputting a value and setting it ran a function which normalized each of the edge weights against the user input and refreshing the graph (Figure 4). All three of these new features were coded before the implementation of the current architecture and were thus moved into the new architecture.

Figures

User

Controller

View



Uses

Manipulates

Updates

Sees

Model

s

**A close up of a map

Description automatically generatedA close up of a map

Description automatically generated**

Figure 2: Medium-scale gene regulatory networks automatically produced from GRNsight. Graph A has its edge weight threshold for edges to be shown as grey set at the default 5 percent of the largest edge weight in the graph. Graph B has the threshold set at 80 percent of the largest edge weight. Graphs remain in a similar location when changing the threshold on the slider when in force graph mode. Changing the threshold does not affect the position of the graph when in grid layout.

Figure 1: A visual representation of the Model-View-Controller paradigm. This software architectural pattern requires the codebase to be contained in three distinct processes that contain specific controls of the application. The user uses a portion of the application, which is heard by the controller, and relayed to the controller. This updates the state of the application, which in turn is heard by the View code, which updates what the user sees.

B

A

A close up of a map

Description automatically generated

Figure 4: This shows a small graph rendered by GRNsight, with different edge weights. The top two graphs have been rendered as is, and the bottom two have been set to use the same normalization factor. The bottom right graph now has an edge thickness rendered by GRNsight that is more accurate to the actual relationship between nodes A and B when compared to the bottom right graph.

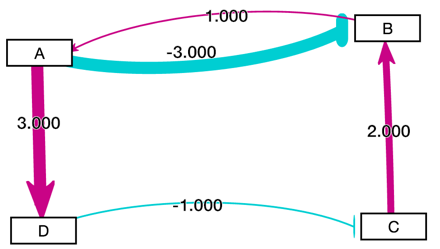
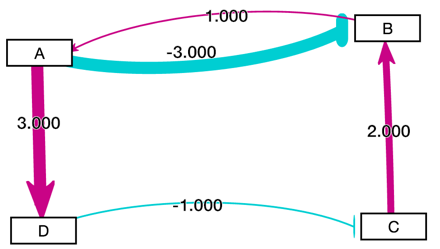
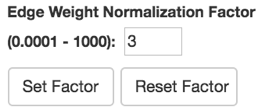
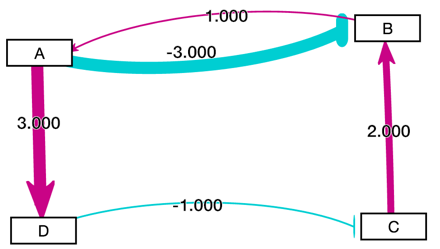
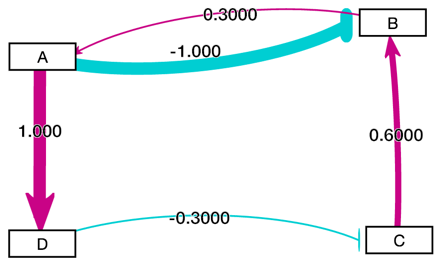
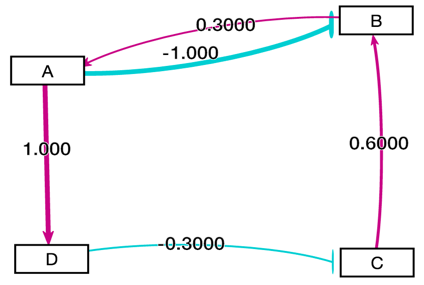


Figure 3: A medium scale gene regulatory network laid out automatically by GRNsight. Gray edges are shown visualized as dashed lines. This feature was added to assist with colorblind users distinguish between miniscule edge weights with an activation or repression relationship, and edges with a negligible effect on nodes/genes.

A close up of a map

Description automatically generated

Figure 5: A view of GRNsight's application interface. 1) File formats button used to import and export Excel, SIF, or GraphML files. Demo files are provided. 2) Grid Layout button allows the users to toggle the graph between a grid layout and a force graph layout. 3) The node coloring menu allows users to modify parameters of the node coloring visualization.Dataset options are automatically generated from expression data sheets detected in an Excel input workbook. 4) Link distance determines the minimum distance between nodes. Nodes have a charge, which repel or attract other nodes. Reset functionality sets all parameters to default. These sliders allow users to modulate this functionality. Locking the parameters prevents any further changes. 5) The viewport is where the user is able to view a graph. The graph bounding box can be separated from viewport. Multiple viewport sizes available. Zooming and scrolling enabled. 6) GRNsight includes options to show or hide the weight values. Radio buttons enable the user to always see edge weights, never see edge weights, or see edge weights upon mouseover of the edges. 7) These buttons allow a user to modulate the viewport by scrolling or zooming through the viewport. 8) This button allows user to set edge weight normalization factor in user interface. Edge thicknesses for different graphs can be rendered on the same scale, so this optionn. facilitates accurate visual comparison. 9) This slider allows the gray edge threshold to be customized. Gray edges allow users to visually gauge whether a particular regulatory relationship is not important. By default, edges are colored gray if the magnitude of its value is <= 5% of the absolute value of the maximum edge weight. As the threshold value increases, only the highest magnitude regulatory relationships are rendered in color. Gray edges can also be rendered as dashed lines to further distinguish the edges.

References

Koppers, T., Ewald, J., Larkin, S., & Kluskens, K. (2015). Webpack: Module bundler. *URL https://github.com/webpack/webpack*.

Leff, A., & Rayfield, J. T. (2001). Web-application development using the model/view/controller design pattern. In *Enterprise Distributed Object Computing Conference, 2001. EDOC'01. Proceedings. Fifth IEEE International* (pp. 118-127). IEEE.

Vipul, A. M., & Sonpatki, P. (2016). *ReactJS by Example - Building Modern Web Applications with React.* Packt Publishing Ltd.