Mihir Samdarshi

March 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**

Introduction

The central model of molecular biology describes how the flow of information in a cell during gene expression goes from DNA to RNA to protein. Transcription factors control gene expression by binding to regulatory DNA sequences. Activators increase gene expression. Repressors decrease gene expression. Transcription factors are themselves proteins encoded by genes. A gene regulatory network (GRN) consists of genes, transcription factors, and the regulatory connections between them, which govern the level of expression of mRNA and proteins from those genes. These gene regulatory networks can be represented in a graph format, with nodes and edges. Each node in a graph represents a regulatory transcription factor, with each edge representing a regulatory relationship, either an activation or repression relationship from the transcription factor to its target gene.

The Dahlquist lab has been investigating the early transcriptional response to cold shock in budding yeast. The lab has measured gene expression using DNA microarrays for the wild type and transcription factor deletion strains. This has led to the identification of gene regulatory networks that have been hypothesized to be important to the regulation of this type of environmental stress response. The Dahlquist lab uses the microarray data as inputs to a mathematical model that models the dynamics over time. This model is known as GRNmap and is a sister project of GRNsight. GRNmap is a differential equations model of the changes in gene expression over time for a gene regulatory network (Dahlquist et al. 2015). Each gene (node) in the network has an equation. The parameters in the model are estimated from laboratory data. The weight parameter, *w*, gives the direction (activation or repression) and magnitude of the regulatory relationship. GRNmap produces an Excel spreadsheet with an adjacency matrix representing the network. 0 represents no relationship between a regulator and its target gene. Initially, 1 represents a regulatory relationship where the gene specified by the column controls the gene specified by the rows. After the parameters have been estimated, a positive weight value indicates activation and a negative weight value indicates repression. The magnitude of the weight defines the strength of the relationship. However, GRNmap does not generate a visual representation of GRNs. Thus, a need for a visualization tool arises.

GRNsight was developed after the Dahlquist lab observed that commonly used graphing software for similar adjacency matrices was primarily optimized for large data sets. Software like yEd (Wiese et al., 2004), Cytoscape (Shannon et al., 2003) and Gephi (Bastian et al., 2009) are common software packages that allow for the visualization of networks. However, these tools are most effective in organizing and displaying large data sets (larger than 100 nodes). This means that the emphasis of the visualization is on important or central nodes. In the Dahlquist lab, the primary interest lies in the relationships between the nodes (or genes) rather than the nodes themselves. Furthermore, yEd is not an open source project; this means that there is no obligation for developers to listen to their primary audience, no opportunity for contributions to the project, and the software cannot be built upon. Gephi, on the other hand is not targeted towards biological applications and thus lack visualization features and ease of use that would make it a valuable addition to a dynamic systems biology laboratory like the Dahlquist lab.

To develop a successful visualization tool, the Dahlquist lab determined that it was essential that GRNsight contain features common to any successful visualization tool for biological pathways. These features are best described by Pavlopoulos et al. (2015) and Saraiya et al. (2005). One of the first requirements is that pathways must be automatically constructed, and not represented statically; for example, this would mean that a pathway constructed in an Excel sheet is not a good representation for visualizing a pathway. Other requirements included being able to gain context for the pathway within the application. Additionally, they stated that the visualizations should be able to provide information attributes that visualizations should reveal through their visual representations (Saraiya et al., 2005). Detailing further, Gostner et al. (2015) state that the aesthetic appearance of the model must promote the the readability and maintainability of models. Finally, many of the reviews of current systems biology visualization tools state that the visualization tool must promote sharing of the data and results throughout the scientific community. This includes features like data interoperability, visual sharing, and ability for collaboration.

GRNsight is optimized for visualizing the relationships in small- to medium-scale gene regulatory networks. GRNsight uses the Data-Driven Documents (D3) JavaScript library to generate a graph derived from input network data. D3 dynamically manipulates HTML and Scalable Vector Graphics (SVG) to form the elements of the graph. GRNsight implements D3’s force layout algorithm which applies a physics-based simulation to the graph; users are able to modify the physics of this simulation. GRNsight then allows users to easily modulate nodes or lay them out automatically in an alphabetized grid. Nodes can also display time course data sets to allow users to better understand the significance of the overall edge relationships over time. Users can also direct attention to certain edge weights by setting a threshold to color edges gray. Comparison of different graphs can also be facilitated by setting a factor against which graph data can be normalized against. GRNsight also allows users to import networks in the commonly used formats such as SIF and GraphML. Such features have enabled GRNsight to become an essential utility in the data processing pipeline utilized by the Dahlquist lab.

GRNsight has been built and revised to its current version over the course of five years. The initial development began in 2014, and since then has had around 8 different developers contribute towards its codebase. GRNsight has consistently tried to follow the Test-Driven-Development best practice that encourages developers to write tests for any potential changes before actually making those changes (Bissi, 2016). However, even given this best practice, the codebase has gotten quite haphazard having changed so many hands and developed fairly rapidly by undergraduate students. This has led to detrimental effects, such as a rapidly increasing number of bugs in the software, correlated with the increasing number of features, along with the codebase equally not as user-friendly.

A central state register was determined to be the best way to resolve the core issues GRNsight was having, namely an increasing number of bugs and vastly disorganized code. It was determined that the Model-View-Controller (MVC) software architectural pattern was the best implementation of this system. MVC was first described to the general public by Krasner & Pope (1988) as a pattern for implementation of user interfaces in 1988. After the rapid rise of web applications in the dawn of the internet, MVC quickly became the software architecture of choice for most use-cases. Leff and Rayfield (2001) described the implementation of a basic web application using this paradigm. It calls for the separation of the central state of the application (the Model) from the implementation of the code that was the basis of what the user saw (View) from the code that handled any particular interactions the user could have with the web application (Controller).

I will describe the major revisions that GRNsight has gone through since its last major release of version 3.0.0 in April of 2018. This includes the addition of new features to make GRNsight more accessible to users, allow for better sharing of graphs, and increase visualization options. Furthermore, I will also describe the implementation of the new MVC architecture for GRNsight and its significance to how GRNsight operates today. I will show that GRNsight is a powerful tool that accomplishes its task of visualizing small- to medium-scale gene regulatory networks in a lightweight, easy-to-use, and feature-rich manner, all while implementing industry-standard best-practice software development practices.

Results

*Codebase*

Since its inception, GRNsight’s client and server code have been written in JavaScript. GRNsight uses Node.js as a server-side runtime environment, with ExpressJS as a framework; it delivers key information to the front-end client code that runs the network visualization. Since 2018, GRNsight has used Webpack in order to attain faster delivery of the service, as well as increased modularity of code. Webpack allows developers to bundle assets, such as images and web elements, 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 ECMAScript 2015 or ES6. This allowed for the use of advanced features such as concise function declarations, as well as exports and imports. Furthermore, the addition of classes gives the GRNsight development team the option of introducing further modularity. However, the conversion to ES6 also necessitated the addition of Babel, a compiler, to convert ES6 and newer specification code into a backwards compatible version of JavaScript in current and older browsers or environments. This ensures the ES6 code in the GRNsight codebase is user compatible across the entire spectrum of our potential users. Additionally, the addition of Babel prepares the GRNsight codebase for the porting to the popular ReactJS framework, as it compiles JSX syntax, a portmanteau of HTML and JavaScript used heavily in ReactJS. Other important packages include D3.js, the main visualization library, ESLint for code formatting, and Chai and Mocha for testing. D3.js is the most commonly used JavaScript library for the creation of visualizations.

*Current Features*

Currently, GRNsight contains a plethora of features that used to help produced better visualization options. Since the graph is rendered with physics enabled on nodes and edges, there is an option to automatically lay out the graph in an alphabetized graph; there are also sliders provided to modify the charge of nodes and increase or decrease the link distance of edges if the graph remains in a force graph layout, as well as lock those sliders. If timecourse expression data is provided in an uploaded worksheet, options are provided to choose which expression data set to display on the top or bottom half of the node; users are also provided options to average the replicate values of the timecourse data and to modify the log fold change max value. One can also change the size of the viewport to three predetermined options or to automatically fit the screen, as well as choose to restrict the graph to the viewport. Inside the viewport, arrows and sliders are provided to move the viewport around, or to zoom in and out of the graph. The user may also use their mouse for those controls. Finally, users may modulate the value against which edge weight magnitudes are normalized against, choose a threshold at which to display an edge weight as gray, or show gray edges as dashed.

*Architecture of GRNsight v. 1.0.0 to GRNsight v. 3.1.3*

The GRNsight project was originally written in the spring of 2014 with the first release coming online on March 20, 2014. The original codebase was less than a third of the size than it currently is. GRNsight version 1.0.0 features were limited to simply importing and displaying a Microsoft Excel workbook with unweighted or weighted edges, very few files contained GRNsight’s codebase. At the end of 2014, the folder that contained GRNsight’s the JavaScript code that runs the web client contained only five files. Only two of those files contained code written by GRNsight’s original developers. The other three files were necessary imports, such as the jQuery library and a package to detect when all images have been loaded. As additions of new front-end features accelerated, a similar structure was retained to the initial file structure of GRNsight, with each new feature getting a new file unless absolutely unecessary. 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 original architecture, all three aspects of the application were contained in singular files (*Fig. 1*). 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 force graph layout function. Upon uploading an appropriate file, the user sees the force graph with default physics enabled. The sidebar displays sliders that can be used to alter the current values of link distance and charge; 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 file structures of GRNsight had the potential to be very problematic. The potential for conflicts between different aspects and features of GRNsight was great. These clashes could lead to bugs when a user selected certain combination of actions to the graph. The number of bugs that arose led to the decision to convert the entire front-end codebase to an architecture following the commonly used Model-View-Controller paradigm. This was performed 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 this basic paradigm, GRNsight also separates the state of the application interface and the logic that controls changes to that application state.

The majority of GRNsight’s user 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 the files are designed to be modular, 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 most of these files. This improves code comprehensibility and allows 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 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 functions 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 is 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 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 refactored 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 the user and updates the central store of the 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 using the D3.js library 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 of the feature set. Additionally, each feature was tested completely to ensure that it worked fully 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 layout physics, as well as the recently added button which allows a user to automatically layout nodes in a grid. These features were heavily inter-mixed 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 straightforward, 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 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 (*Fig. 5*). 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 (*Fig. 8*). Finally, another feature which I worked on allowed users to normalize edge weight visualization against a user-inputted value, instead of the default largest magnitude 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 (*Fig. 6*). All three of these new features were coded before the implementation of the current architecture and were thus moved into the new architecture.

Discussion

*Addition of new features*

The newly centralized architecture of GRNsight allows developers to add new features in a highly intuitive and modular fashion. The way that the Model-View-Controller is implemented has a profound impact on how easy the code is to develop, maintain and reuse (Hansen & Fossum, 2005). Thus, the way that MVC was implemented in GRNsight was made such that harmful interactions were minimized, and that minimal code was situated in locations extraneous to the MVC architecture. Most code that faces the client-side is situated in three files, which constitute each component of MVC. Thus, adding new features simply requires the addition of the HTML into the static template files, creation of a handler for the feature into setup-handlers.js, addition of a new application state to grnstate.js, and addition of code to execute the user’s input and update the view to update-app.js. Furthermore, the entire separation of Model, View, and Controllers enable a developer to create pluggable views/controllers; in other words, the complete separation of all components allows a user to change the functionality or appearance of a different component of a feature without the necessity to completely refactor all code associated with a feature due to its appearance completely in one function. However, in the same vein, some modularity can also be hindered by the strict MVC architecture. It is significantly harder to reuse view and controller code, because both are tied to the specific model. Thus, a developer must take extra and often superfluous steps to ensure full code compatibility if there are any changes to the model, or in order to ensure reusability of model or controller code (Veit & Hermann, 2003).

*Central Store for Application State*

The addition of a central state for GRNsight in the form of an object creates a cross-reference that is easily accessible by both the code itself, and developers (Hansen & Fossum, 2005). This central state, in the form of an object in GRNsight, stores most of the information regarding the application state, save for the position and physics of the actual nodes and edges displayed in the simulation. There are many advantages to having this central store of information. First, having a central state store prevents unintended state conflicts. Conflicts between application sub-states often create bugs that, especially in front-end code, may affect a user’s experience. According to Ceaparu et al. (2004), errors in an application are a leading cause for end-user frustration. A few bugs that were present under GRNsight’s previous architecture were immediately removed with the conversion of the architecture to MVC. Some of these bugs occurred when a user changed the normalization factor of a graph’s edges while a graph was in a grid layout configuration. Under the previous architecture, the graph would revert back to a force graph layout. This was resolved after the transition to the MVC architecture, as the separate controllers for these features referenced different parameters within the central state store. However, when bugs do occur under the MVC architecture, having a central state store makes diagnosis of the bugs straightforward. After transition of the application to the new architecture, a bug arose where, even upon clicking on the “Lock Force Graph Parameters” check box, the Force Graph Parameter sliders would not be locked. Diagnosing and creating a solution involved simply looking for all functions that relied on that parameter of the central state store, finding the conflict, and removing the conflict.

Finally, having a central state store allows ease of use for future development (Leff & Rayfield, 2001). In addition to allowing for minimization of potential state conflicts that could arise from new features, the central state allows the developer to access the code more easily and add new modules. Under the previous architecture, blocks of code that might have controlled the same feature were interspersed throughout various files or in different parts of the same file. Under MVC, this code is now guaranteed to be contained between three files, and functions are organized according to the specific feature that they control or interact with. Functions that are shared between multiple functions are located as close to the top of the file as possible. Furthermore, the transition of GRNsight to ECMAScript 2015 (ES6) allows for the importation of code from different locations, allowing code to be much more organized and increasing potential modularity. Components of the MVC architecture are separated into modules located in separate files because ES6 allows each file to nonetheless interact with each other. Webpack additionally allows for this modularity to continue to work across all platforms, including browsers that may not yet support the new ECMAScript featureset.

*New Features*

The three modifications to visualizations that were added allow users expanded functionality to modulate their graphs and enhance overall user experience and accessibility. These additional features allow users to set a threshold at which to visualize edges as gray or colored, view those gray edges as dashed lines, and to allow users to normalize edge weight visualization against a user inputted value.

Allowing users to set a threshold at which to visualize gray edges allows users the opportunity to display edges of importance. Users control this feature via a slider that corresponded to a percentage of the largest magnitude edge weight. By default, the visualization factor is set such that edges with an edge weight magnitude of less than 5% of the largest magnitude edge weight are visualized as gray. Movement of the slider increases the percentage by one percent increments, up to a maximum of a hundred percent and a minimum of one percent. The default of five percent was chosen as these relationships tend to be fairly insignificant in a small- or medium-scale gene regulatory network. However, this feature was added because allows users to be able to decide which edges are significant and share those with peers. Since edges are colored blue and red to correspond to activation or repression relationships, setting an edge weight visualization percentage allows viewers to pay attention to those colored edges rather than the thin, gray relationships.

The addition of the feature that allowed users to control the visualization of gray edge weights was motivated primarily because a member of the LMU Bioinformatics and Biomathematics research groups was color-blind. The color-blind member of the research group pointed out that it was difficult for him to distinguish between thin magenta/red edges and gray edges. Red-green color blindness affects approximately 8% of Caucasian males, 5% of Asian males, and 4% of African males worldwide (Birch, 2012); in order to make GRNsight available to a larger portion of the scientific community, solutions were considered to allow for more accessible viewing options. Two solutions were ultimately chosen; the first to change the edge weight coloring from cyan and magenta to blue and red, and the second to display gray edges as dotted lines, so that even without color users would be easily able to distinguish thin edges from gray edges (Okabe & Ito, 2002).

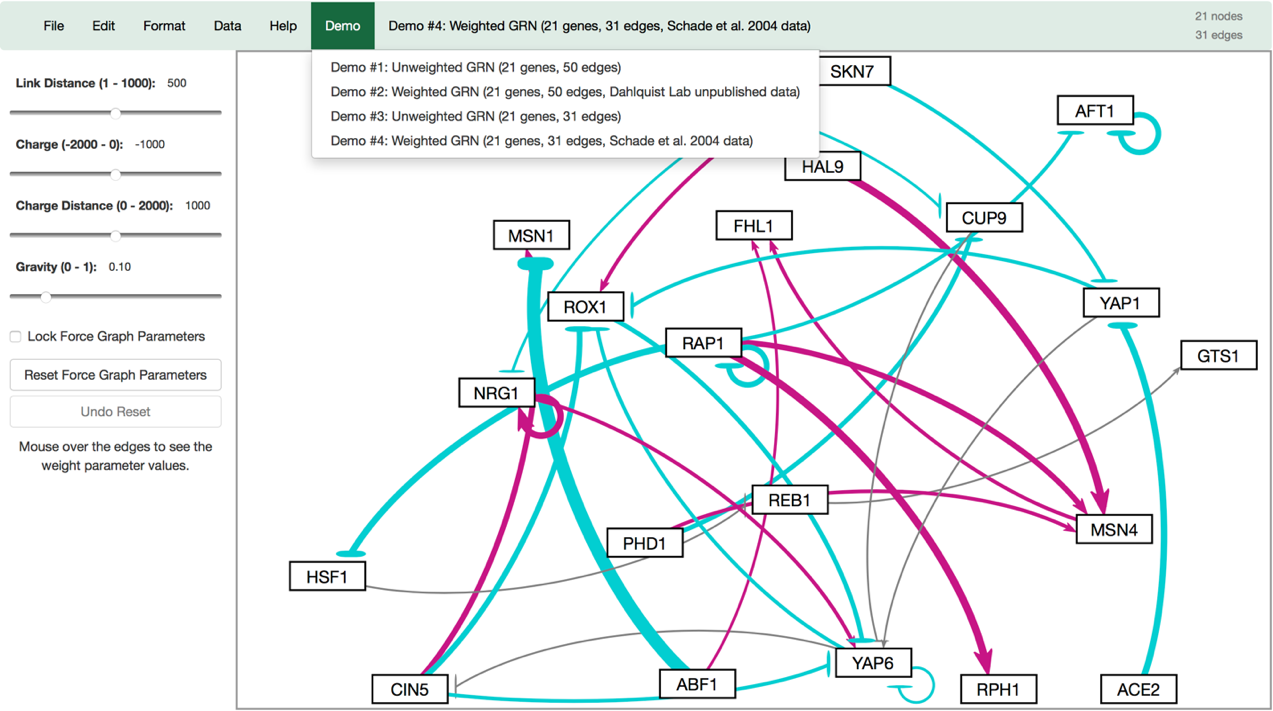
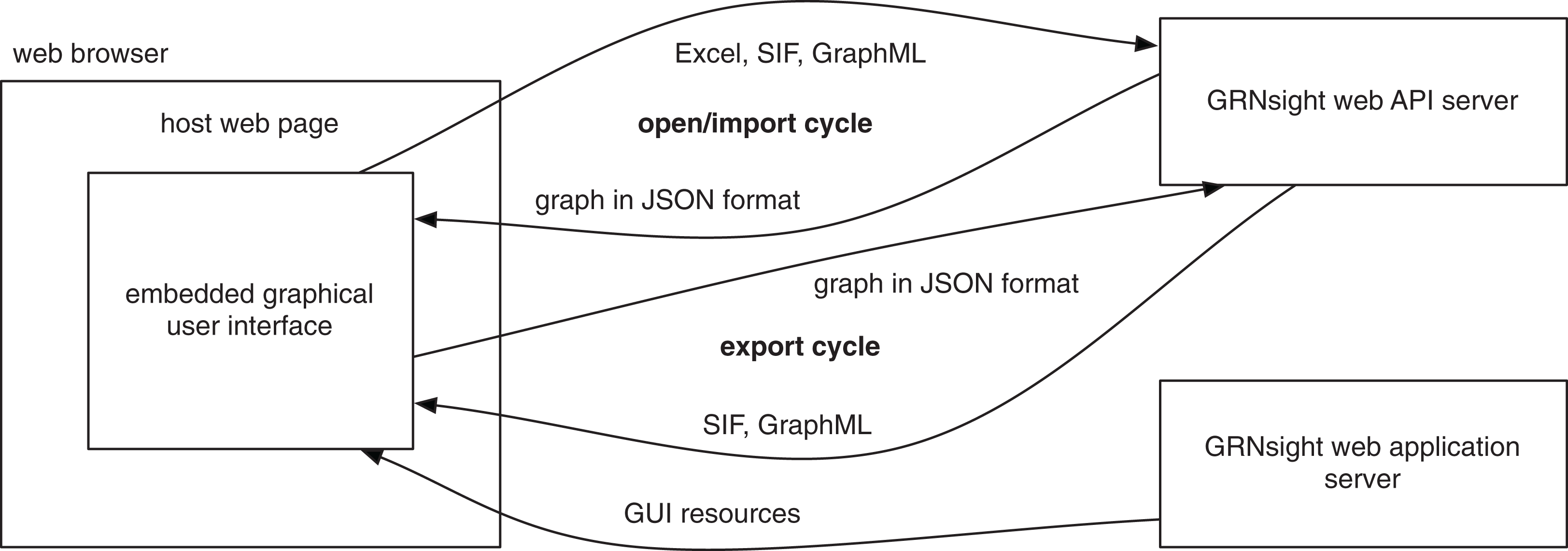
Finally, the last feature added in to improve GRNsight’s functionality was the option to set an edge weight normalization factor. In order to work within d3.js’s visualization limitations, edges are grouped into twelve “bins”, corresponding to edge thickness. The edge with the largest weight magnitude is placed in the being placed in the “bin” that corresponds to the largest possible edge thickness; edges are then assigned to subsequent bins based on what percentage of the largest weight their weight magnitudes are However, testing within the Dahlquist lab revealed that biologists are often trying to compare graphs of multiple gene regulatory networks in which the strongest magnitude relationship may not always be the same. Thus, the original programming of GRNsight did not facilitate accurate comparison of graphs. Setting an edge thickness normalization to a certain parameter allows users to compare graphs with differing maximum edge weight magnitudes. Thus, research that relies on the comparison of different gene regulatory networks can be easily shared.

*Future Directions*

With GRNsight now a much more modular, developer-friendly, and user-friendly software following the refactor to MVC and addition of new features, there are many possible directions that the GRNsight team could take with our software. The recent addition of the Cytoscape.js library (Franz et al., 2015) allows the potential addition of useful graph statistics. These statistics include global clustering coefficient, betweenness centrality, and degree distribution. Statistics like these allow scientists to visualize important or insignificant relationships between genes (nodes) and to determine important properties regarding significance of certain nodes or graphs. Systems biology analysis tools such as InnateDB (Breuer et al., 2012) and ConsensusPathDB are two examples of software that have already successfully implemented the functions that enable graph statistics in Cytoscape.js. Additionally, the transition of test architecture to better follow the new web-client architecture would allow GRNsight to retain a more intuitive testing process and would help to better attain bug-free software. This transition would also go hand in hand with the addition of tests specific to the new architecture in order to help GRNsight return to the Test-driven development (Beck, 2003) philosophy, which it has strayed from in recent history. Finally, in order to better increase GRNsight’s share-ability, a feature could be added to allow export of a graph visualization to an easily shareable format such as Portable Network Graphics (PNG) or Scalable Vector Graphics (SVG).

*Conclusion*

GRNsight has undergone many changes since its inception in 2014; however, it has retained the same architecture for the majority of its lifetime. During the course of this project, a major effort was undergone in order to convert GRNsight to a more sustainable and prudent architecture, termed Model-View-Controller. This architecture allows GRNsight to contain more modularity, better developer access, and less bugs. Furthermore, the addition of new features that allow users to better modulate the network graphs created by GRNsight allow users better options and more flexibility. Finally, preparation for a move to the ReactJS framework allows GRNsight to be more viable in the long-term.

****Figures

**activation**

**repression**

**weak influence**

**Key:**



**gene**

Figure 2: A view of GRNsight's application interface at the release of version 1.0.

Figure 1: A representation of GRNsight's version 1.0 architecture. Notably, there is no emphasis on maintaining a Model-View-Controller paradigm to reduce application instability.

s

A close up of a map

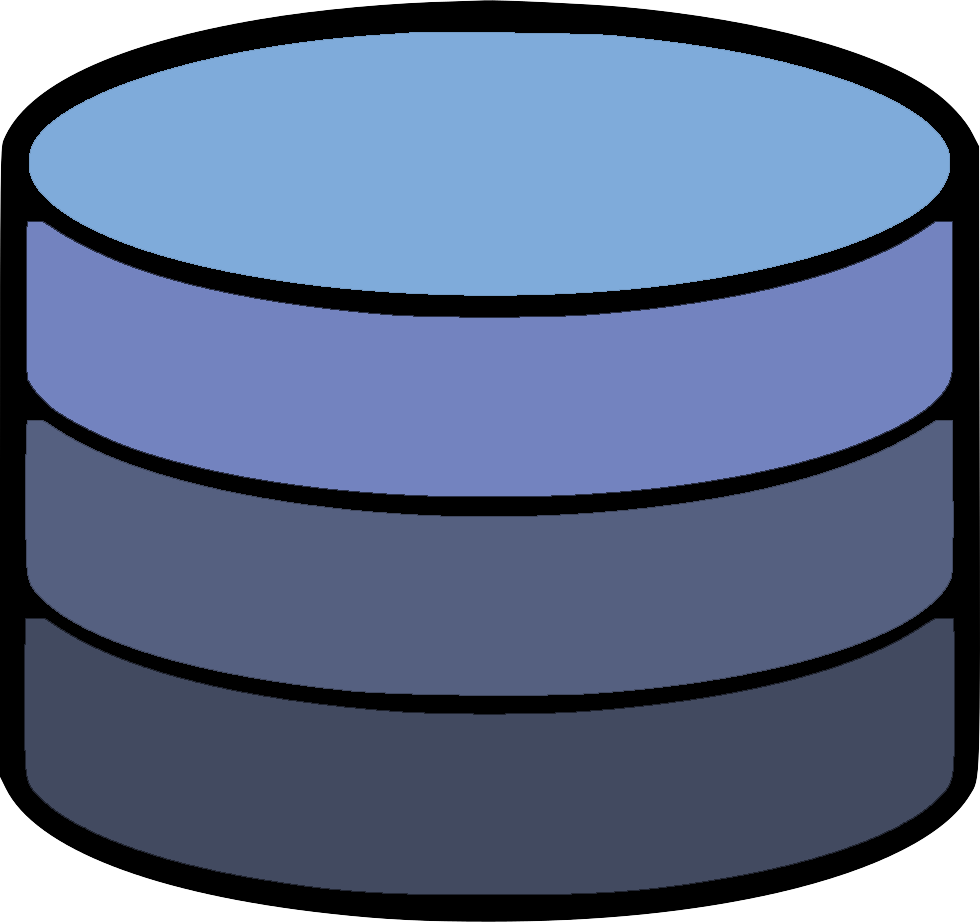
Description automatically generated

Figure 3: A view of what GRNsight v. 3.1 looks like with the addition of all the new features. This is in comparison to the features present in GRNsight v 1.0, seen in Figure 2.

User

Controller

View



Uses

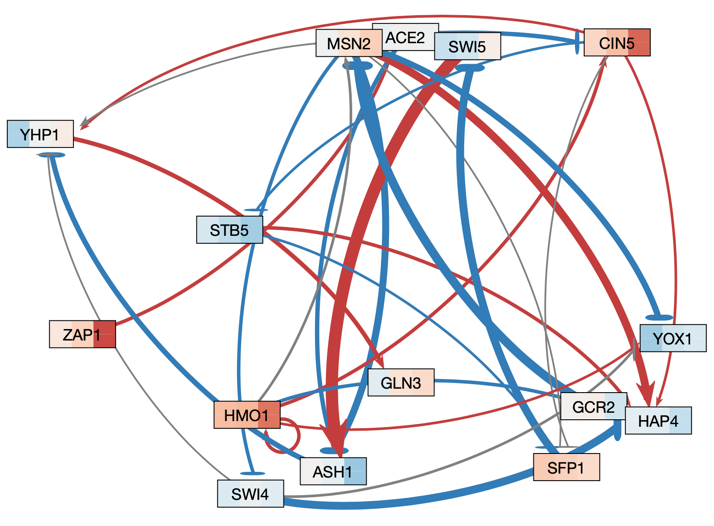
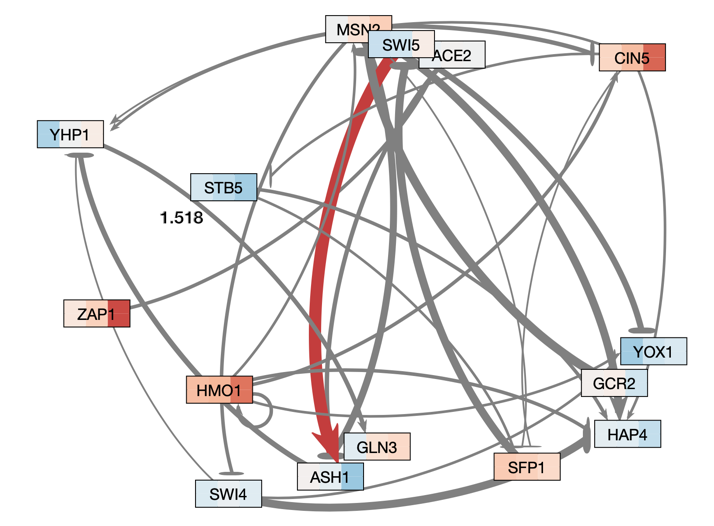
Manipulates

Updates

Sees

Model

Figure 4: 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.

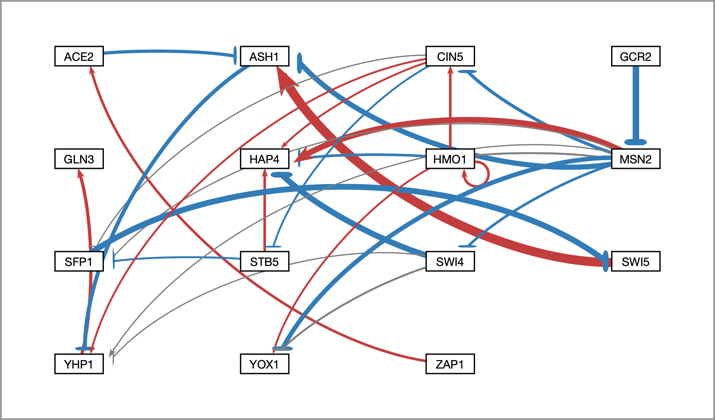
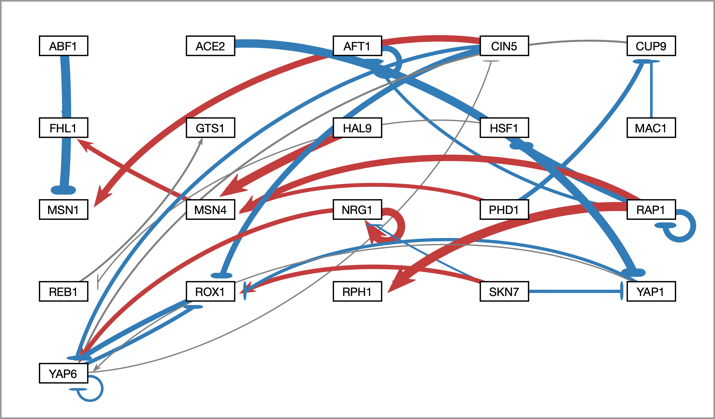


A

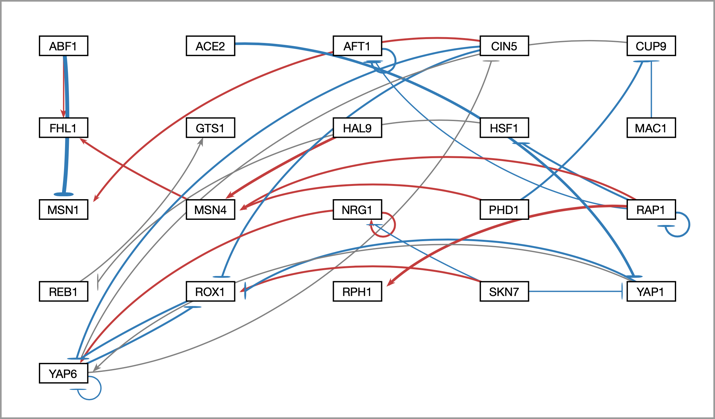
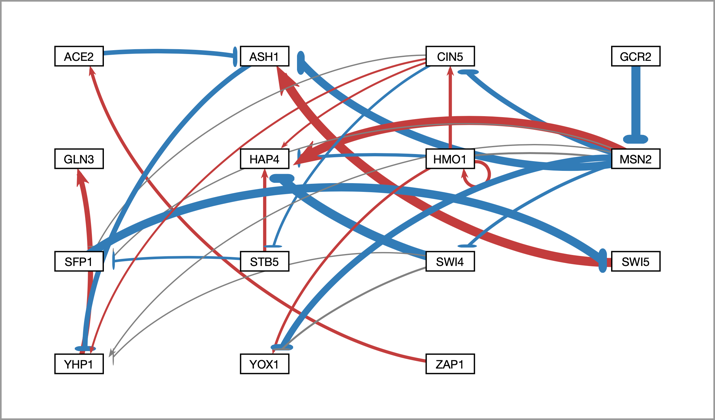
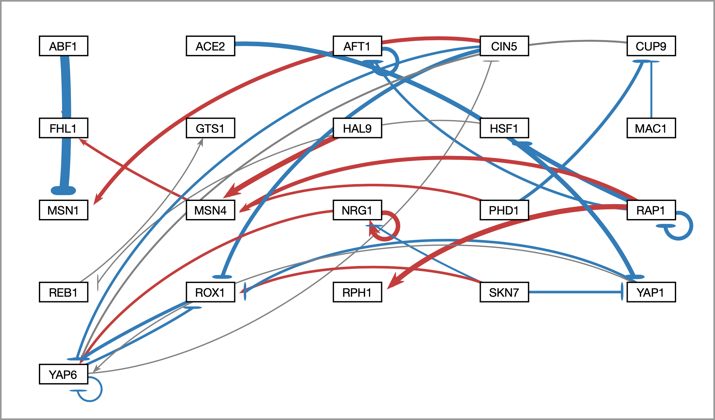
B

Figure 5: 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.

**Normalization Factor = 1.5**



**Normalization Factor = 5.21**



**Normalization Factor = 3**

**Normalization Factor = 3**

**Normalization Factor = 10**

**Normalization Factor = 10**

Figure 6: This shows a small graph rendered by GRNsight, with different edge weights. The top two graphs have been rendered as is, and the bottom four 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.

A close up of a map

Description automatically generated

**Gray Edge Threshold = 50%**

**Gray Edge Threshold = 5%**

Figure 7: To allow users to direct attention to certain nodes, users may adjust gray edge threshold. The graph on the left shows a graph with the gray threshold set to 5% of the largest magnitude edge weight, and the graph on the right has a threshold set to 50%

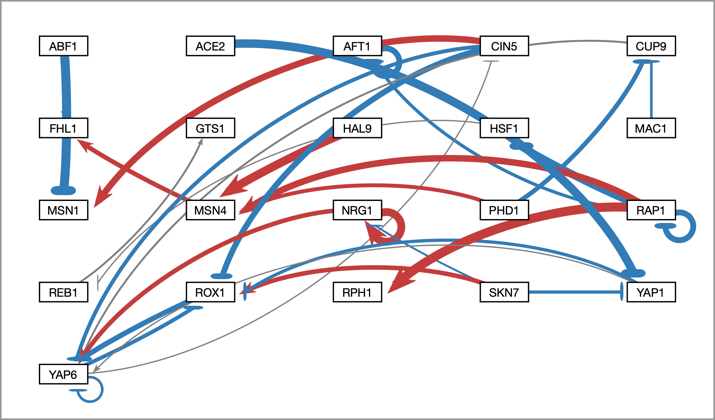
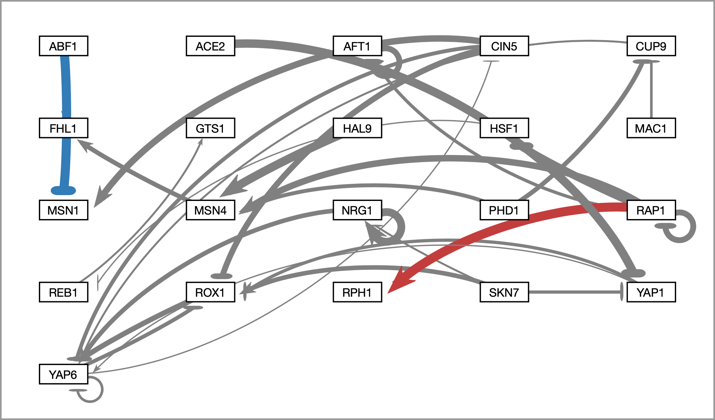
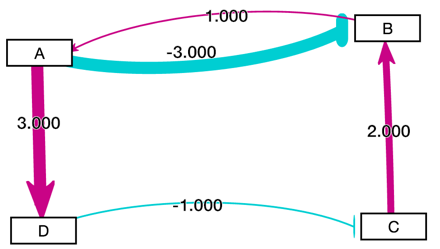


Figure 8: A medium scale gene regulatory network laid out automatically by GRNsight. Gray edges are shown visualized as dashed lines. This feature was added to assist with colorblind users distinguish between miniscule edge weights with an activation or repression relationship, and edges with a negligible effect on nodes/genes.



References

Bastian, M., Heymann, S., & Jacomy, M. (2009, March). Gephi: an open source software for exploring and manipulating networks. *In Third international AAAI conference on weblogs and social media*.

Beck, K. (2003). *Test-driven development: by example*. Addison-Wesley Professional.

Birch, J. (2012). Worldwide prevalence of red-green color deficiency. *Journal of the Optical Society of America A*, 29(3), 313-320. DOI: 10.1364/JOSAA.29.000313

Breuer, K., Foroushani, A. K., Laird, M. R., Chen, C., Sribnaia, A., Lo, R., Winsor, G. L., Hancock, R. E. W., Brinkman, F. S. L., Lynn, D. J.; InnateDB: systems biology of innate immunity and beyond—recent updates and continuing curation, *Nucleic Acids Research*, Volume 41, Issue D1, 1 January 2013, Pages D1228–D1233. DOI: 10.1093/nar/gks1147

Ceaparu, I., Lazar, J., Bessiere, K., Robinson, J., & Shneiderman, B. (2004). Determining causes and severity of end-user frustration. *International Journal of Human-Computer Interaction*, *17*(3), 333-356. DOI: 10.1207/s15327590ijhc1703\_3

Dahlquist, K. D., Dionisio, J. D. N., Fitzpatrick, B. G., Anguiano, N. A., Varshneya, A., Southwick, B. J., & Samdarshi, M. (2016). GRNsight: a web application and service for visualizing models of small-to medium-scale gene regulatory networks. PeerJ Computer Science, 2, e85. doi: 10.7717/peerj-cs.85

Franz, M., Lopes, C. T., Huck, G., Dong, Y., Sumer, O., & Bader, G. D. (2015). Cytoscape. js: a graph theory library for visualisation and analysis. *Bioinformatics*, 32(2), 309-311. DOI: 10.1093/bioinformatics/btv557

Gostner, R., Baldacci, B., Morine, M. J., & Priami, C. (2015). Graphical modeling tools for systems biology. *ACM Computing Surveys (CSUR)*, *47*(2), 16. doi: 10.1145/2633461

Hansen, S., & Fossum, T. V. (2005). Refactoring model-view-controller. *Journal of Computing Sciences in Colleges*, *21*(1), 120-129.

Kamburov, A., Stelzl, U., Lehrach, H., & Herwig, R. (2012). The ConsensusPathDB interaction database: 2013 update. *Nucleic Acids Research*, 41(D1), D793-D800. DOI: 10.1093/nar/gks1055

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

Krasner, G. E., & Pope, S. T. (1988). A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, *1*(3), 26-49.

Leff, A., & Rayfield, J. T. (2001). Web-application development using the model/view/controller design pattern. In *Proceedings fifth ieee international enterprise distributed object computing conference* (pp. 118-127). IEEE.

Okabe, M., & Ito, K. (2002). How to make figures and presentations that are friendly to color blind people. University of Tokyo.

Pavlopoulos, G. A., Wegener, A. L., & Schneider, R. (2008). A survey of visualization tools for biological network analysis. *Biodata mining*, 1(1), 12. doi: 10.1186/1756-0381-1-12

Saraiya, P., North, C., & Duca, K. (2005). Visualizing biological pathways: requirements analysis, systems evaluation and research agenda. *Information Visualization*, *4*(3), 191-205. doi: 10.1057/palgrave.ivs.9500102

Shannon, P., Markiel, A., Ozier, O., Baliga, N. S., Wang, J. T., Ramage, D., ... & Ideker, T. (2003). Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome research*, *13*(11), 2498-2504. doi: 10.1101/gr.1239303

Veit, M., & Herrmann, S. (2003, March). Model-view-controller and object teams: A perfect match of paradigms. In *Proceedings of the 2nd international conference on Aspect-oriented software development* (pp. 140-149). ACM. DOI: 10.1145/643603.643618

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

Wiese, R., Eiglsperger, M., & Kaufmann, M. (2004). yfiles—visualization and automatic layout of graphs. In *Graph Drawing Software* (pp. 173-191). Springer, Berlin, Heidelberg