Anindita Varshneya

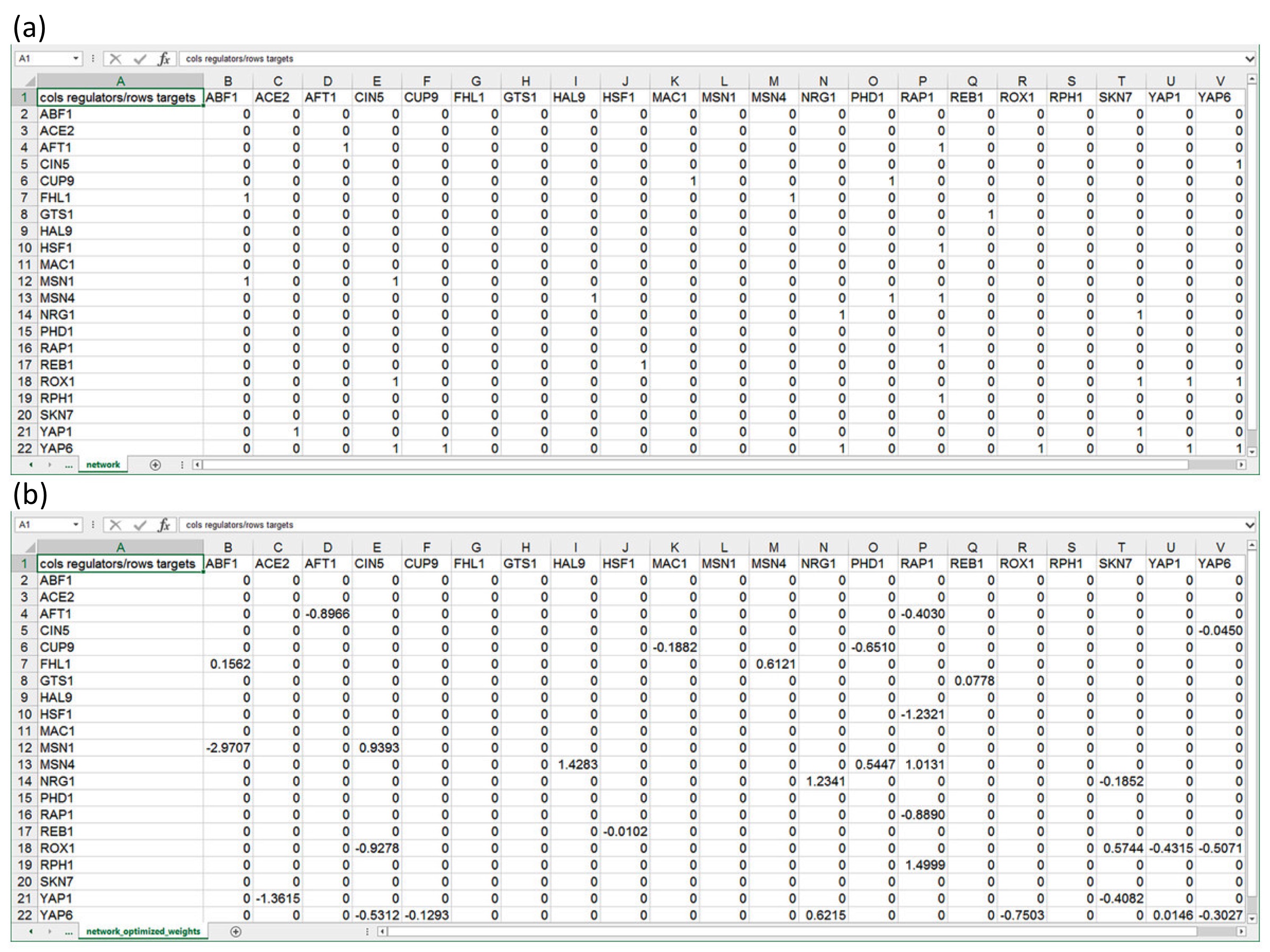
**Test-driven development and new features improve GRNsight: a web application and service for visualizing small- to medium-scale gene regulatory networks**

First Draft

GRNsight is a web application and service for visualizing small- to medium-scale gene regulatory networks (GRNs) (Dahlquist et al, 2016). The central dogma of molecular biology is the process of transcribing DNA into RNA, and then translating RNA into protein. Because proteins are the main functional unit within a cell, regulating the production of proteins is essential to proper cell function. Regulatory transcription factors (TFs) are the molecules within a cell that determine which genes in the DNA should be transcribed into RNA at a particular time. Regulatory TFs can either activate or repress the expression of a gene by attaching at a site prior to the beginning of the gene on DNA. However, the number of TFs in a cell is significantly fewer than the number of genes in the DNA that are regulated by these TFs. To overcome this problem, TFs often use a process called combinatorial controls, where multiple different TFs bind together to collectively activate or repress the expression of a gene. Furthermore, TFs are themselves proteins, so the expression of the RNA that creates TFs is controlled by other TFs.

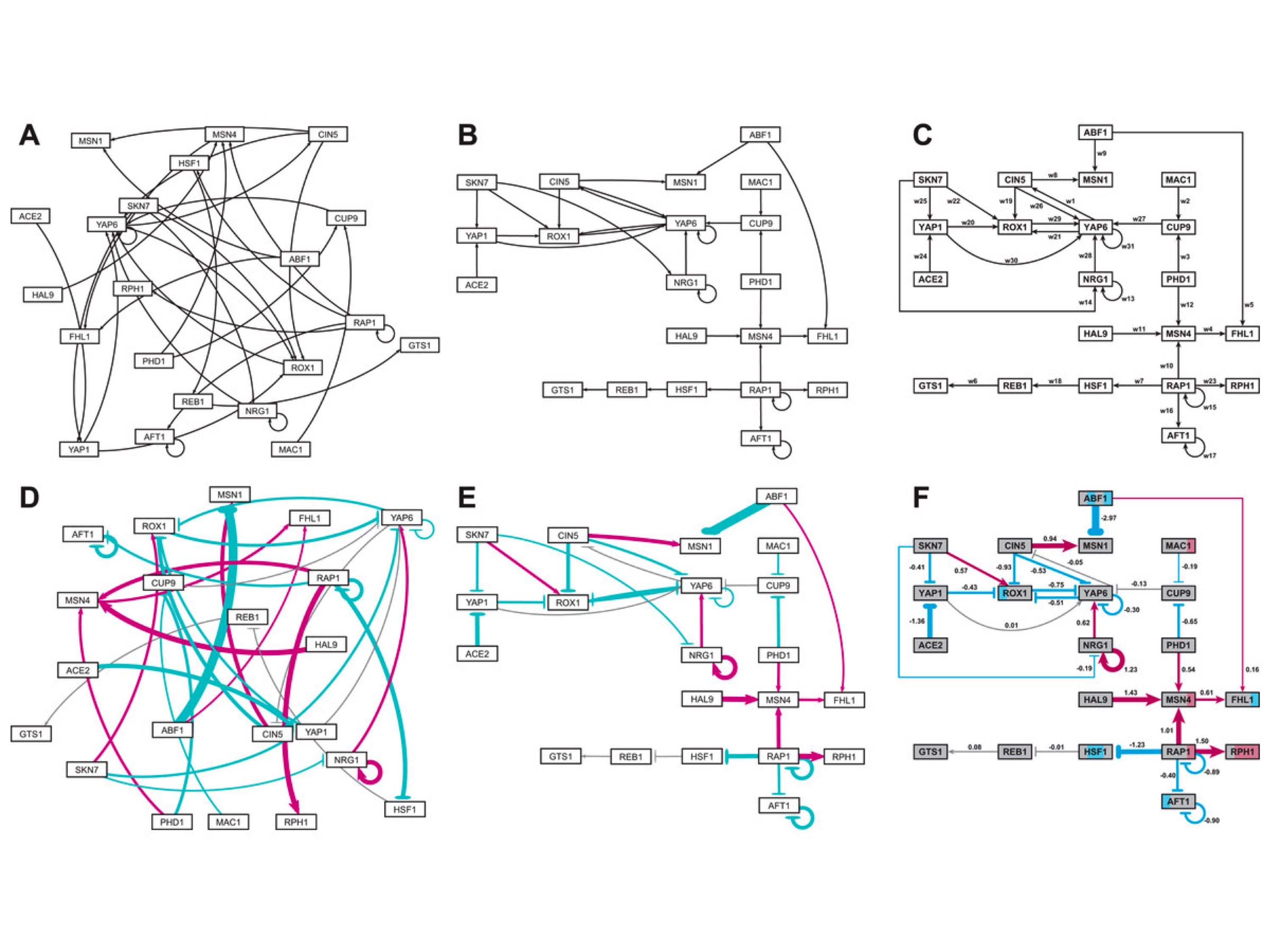
A GRN consists of a set of regulatory transcription factors and the relationships between them. (MacNeil et al., 2011). These interactions define the types and amounts of mRNA and protein produced by the cell. Using wet lab data at an initial set of time points, our sister project, GRNmap, performs parameter estimation and forward simulation of GRNs to predict how TFs might interact with each other at future time points. GRNmap takes in an Excel file containing various sheets of necessary information to complete the estimation, and returns another Excel file that includes the user provided data as well as an additional sheet with estimated data reflecting what might have happened if the experiment continued.

One common presentation of GRNs is in adjacency matrices, which include source genes and target genes listed across columns and rows respectively, and the elements of the matrix indicate the relationship between these genes (Figure 2). GRNmap input files include unweighted adjacency matrices in which a “0” indicates no relationship between the corresponding genes, and a “1” indicates a regulatory relationship of some unknown magnitude and direction. Using this data, GRNmap outputs a weighted adjacency matrix where each “1” is replaced by a more descriptive number which indicates the magnitude of the relationship between the gene and its transcription factor, and the sign of the number (positive or negative) indicates the direction of the relationship (activation or repression).



**Figure 2.** Properly formatted adjacency matrices that can be opened in GRNsight to construct network visualizations. All matrices are directed. Gene names in column headings represent regulators, and genes in row headings represent target genes. (a) An unweighted adjacency matrix where ‘0’ represents no regulatory relationship between the regulator and the target gene, and ‘1’ represents some regulatory relationship between the regulator and the target gene. (b) A weighted adjacency matrix where all ‘1’s were replaced with real numbers that indicate the direction and magnitude of the regulatory relationship between the corresponding genes. A positive number indicates that the regulator is an activator, while a negative number indicates that the regulator is a repressor. (Dahlquist, 2016)

In order to most effectively and efficiently analyze biological data of this type, network visualizations are often used to demonstrate relationships between GRNs (Pastrello et al., 2013). These visualizations are often shown as graphs where different nodes represent genes, and the edges between them represent their relationships. Manually creating these types of visualizations can be tedious and time consuming, so graph visualization software exist to streamline the analysis portion of GRN related research projects (Figure 1). Currently, several different programs exist that visualize network interactions including Cytoscape, Gephi, and yED. While each of these programs effectively diagram GRN visualizations, none of them address the niche cases for which GRNsight was created.



**Figure 1.** Medium-scale gene regulatory networks. (a) and (d) are unweighted and weighted gene regulatory networks, respectively, automatically produced on GRNsight within about 10 milliseconds. In the weighted network, magenta edges represent activating relationships, while cyan edges indicate repressing relationships. Grey edges indicate no substantial activation or repression relationship. The thickness of the lines in these networks indicates the magnitude of the relationship between the nodes. (b) and (e) are user manipulated unweighted and weighted networks, respectively. These graphs were produced in about 10 milliseconds, and took about 5 minutes of manual adjustment to create. (c) and (f) are unweighted and weighted, respectively, gene regulatory networks created on Adobe Illustrator. Each of these networks took several hours to produce. Node coloring in (f) was produced according to the software GenMAPP 2 (Salomonis et al., 2007), which shows expression levels of the gene at different time points throughout the course of the study. (Dahlquist, 2016)

Cytoscape is one of the best-known GRN graphing programs, and has tools that allow users to view and create large networks (Shannon et al., 2003). While Cytoscape is compatible with all of the file types GRNsight is also compatible with, Cytoscape does not take in adjacency matrices as a data structure, and is therefore not compatible with GRNmap generated output sheets. In fact, because adjacency matrices are the most common format through which GRN data is presented, many users uploading GRN data produced in microarrays will need to reformat their data in order to successfully use Cytoscape. Furthermore, Cytoscape is a stand-alone program that a user must install onto their machine. While this may not always raise issues, in the context of a university-based lab, installing new software onto a university machine is often difficult and requires admin login information that is not available to students. Because Cytoscape can run as a stand-alone application, it comes with several options and features that allow users to customize their visualizations and get more information about their networks than other web-based programs. While this is beneficial in some cases, this complicates the learnability of the software, so it is not the best tool for novice users and researchers. Finally, Cytoscape is optimized to present visualizations for large-scale networks. While this is useful for understanding genome-level interactions, it is not as useful when analyzing particular processes, which produce small- to medium-scale networks.

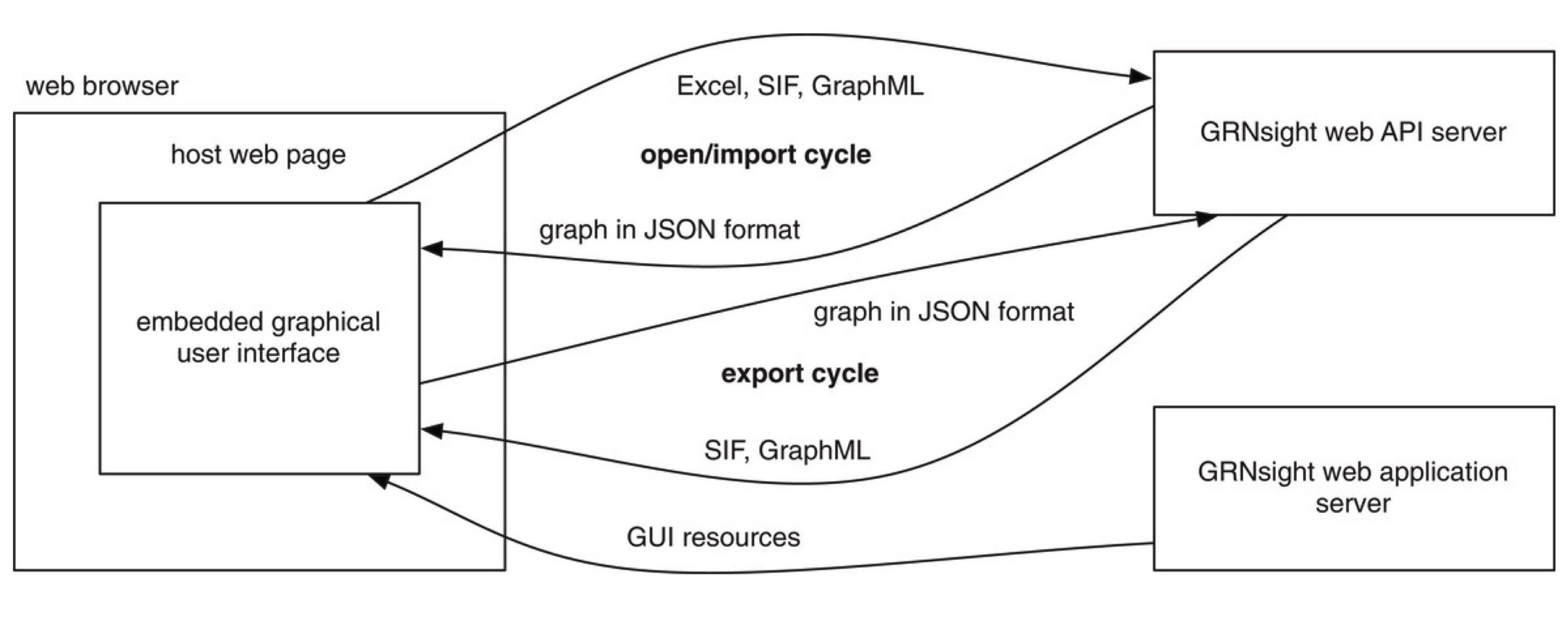
Gephi is another popular software aimed at visualization and explanation of large-scale networks (Bastian et al., 2009). While Gephi boasts its learnability and user interface, it still faces several of the problems previously addressed in regards to Cytoscape. Gephi also needs to be installed onto a user’s machine in order to be accessed, and therefore does not address the niche target audience that GRNsight is tailored towards. Despite simplifying the program and making it more accessible to the average user, Gephi still provides several customization options that would only be beneficial when understanding large-scale networks. Ultimately, like Cytoscape, Gephi was created for researchers analyzing large amounts of data in large-scale networks, which does not address the key concern that led to the development of GRNsight.

yEd is another GRN visualization software that allows users to upload Excel files or XML files containing adjacency matrices and produce directed graphs of that data (Wiese et al., 2004). This program allows users to customize the type of graph created (flowchart, family tree, semantic network, etc.) and also allows users to arrange and export their graphs easily. Unlike the previously mentioned software, yEd has a browser-enabled version of the application in addition to the stand-alone version that must be installed onto a users machine. However, the online version of the program is limited to only a certain number of nodes and edges that would prevent diagramming of some medium-scale networks. In order to access the full version of the program, the user must install the program onto their machine and pay for the program. While yEd comes closest to addressing the chief concerns that led to the development of GRNsight, it does not provide a solution for each of the goals our research team had for creating GRN visualizations.

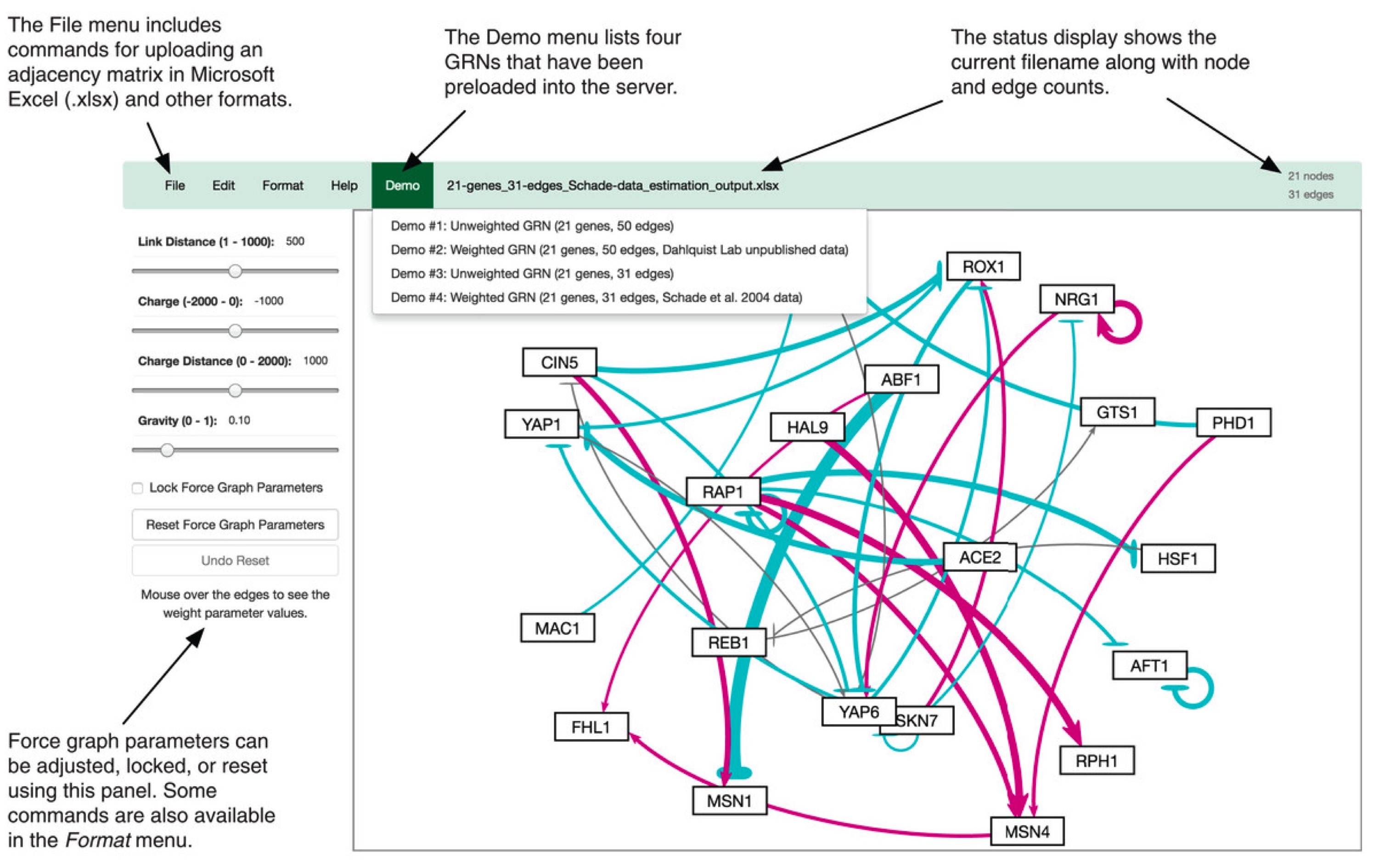
In order to address the concerns that none of the previously mentioned programs adequately solved, the Dahlquist lab created GRNsight with the following guidelines (Dahlquist et al., 2016):

* Exist as a web application that does not require installation.
* Have high learnability so that any novice user could quickly understand the program.
* Accept Excel files containing weighted or unweighted adjacency matrices similar to those produced by GRNmap.
* Automatically produce small- to medium-scale, weighted or unweighted, directed network graphs that allow for easy and efficient analysis of biological data.

GRNsight effectively addresses these goals while also following open-source and software development best practices. GRNsight exists as a free, open source web application that accepts Excel files containing properly formatted adjacency matrices, GraphML data, and SIF data in the production of GRN visualizations. The program uses a sophisticated server-oriented architecture where user provided data is sent to our GRNsight web API server to be parsed into a JSON object that is ultimately sent back to the web browser to be presented as a graph visualization. The user can then choose to export the data in SIF or GraphML formats, which sends the JSON data back to the web API server where is it converted into the requested data type (Figure 3). Once uploaded to the web browser, visualizations can be customized using Force Graph Parameters that sit on the left-hand side of the screen (Figure 4). Additionally, nodes have drag and drop properties that allow users to manipulate graphs and adjust them to their liking. Double clicking on a node releases it from its “dropped” location, and allows the node to respond to any customizations the user might make in the Force Graph Parameters.



**Figure 3.** GRNsight has a complex, server-oriented architecture. The web browser holds the web page and the embedded graphical user interface. When a network spreadsheet is opened on GRNsight, the GRNsight web API server parses the spreadsheet and sends it back to the web browser in JSON format, where it is interpreted and visualized. If a graph is exported from GRNsight, it is returned to our GRNsight web API server in JSON format, converted into the desired file format, and returned to the user interface. (Dahlquist, 2016).



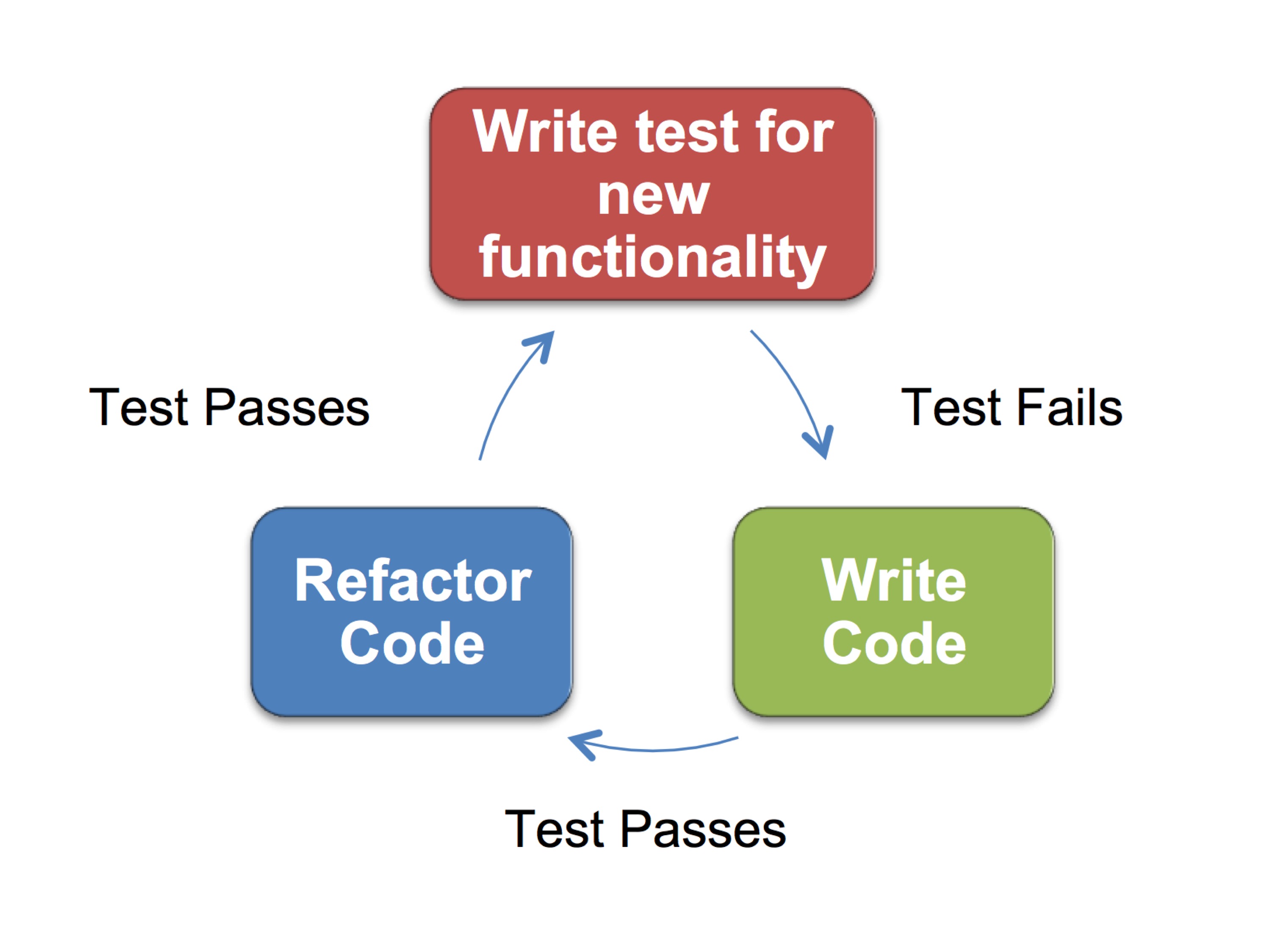
**Figure 4.** An annotated screenshot of the GRNsight user interface. The file menu at the top left of the screen provides commands for uploading an adjacency matrix in Microsoft Excel (.xlsx) and other formats. The demo menu provides four sample adjacency matrices preuploaded on the server. The status bar allows users to see which sheet is currently open in the viewport, and how many genes and edges are in that gene. Force graph parameters on the left-hand side of the screen can be adjusted, locked, or reset according to user’s preference. (Dahlquist, 2016).

Most recently, improvements to GRNsight include the integration of Test-Driven Development (TDD), descriptive error and warning messages, and other front-end options that allow users to customize the appearance of their network.

I. Testing Framework

While the errors and warnings modal is a great tool to assist users in the production of their graphs, these modals are only as useful as they are accurate. In order to ensure that GRNsight is running as expected, and that all relevant errors and warnings are caught accurately, the GRNsight team implemented Test Driven Development (TDD) to GRNsight. TDD is a software development best practice that encourages the developer to make changes incrementally to their program to ensure that it is running expectedly using unit tests (Bissi, 2016). More explicitly, a unit test ensures proper behavior for a particular unit of code. Often, a single new functionality can issue tens of unit tests because each step of that functionality is tested separately.

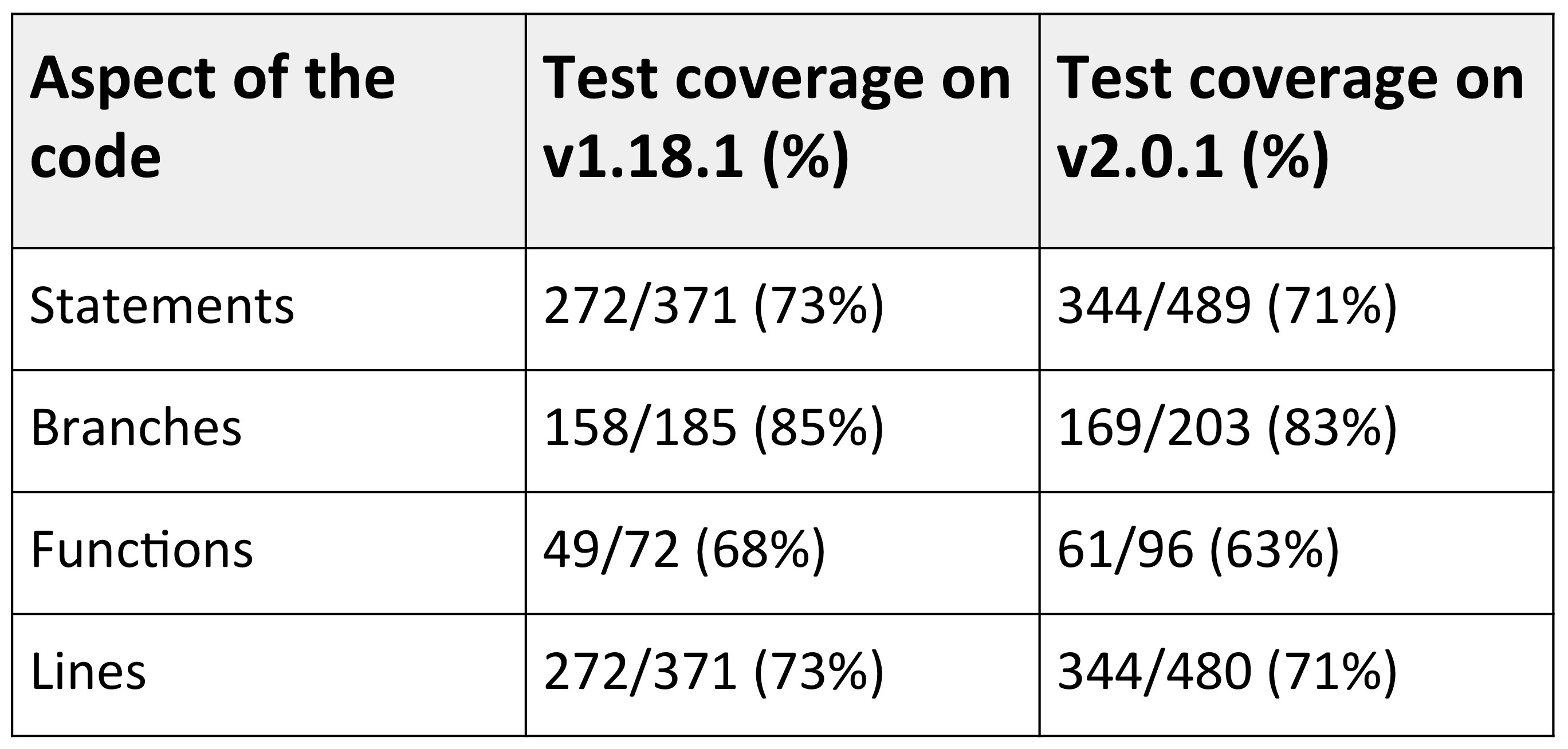
The TDD process begins by writing a unit test for new functionality before coding that functionality into the program itself (Figure 6). This means, at first, the test will fail. Then, the developer should proceed to write code for that functionality. If the new code was written according to predetermined expectations as outlined by the unit test, the test should pass. At this point, the developer can go back and refactor their code to ensure it runs efficiently and is easily understandable. This process is especially important for academic research projects, such as GRNsight, because it ensures that we maintain consistent levels of internal software quality. (Bissi, 2016).



**Figure 6.** The test driven development (TDD) process. The process begins with writing a test for new functionality prior to writing the code for that functionality. Initially, the test will fail. Then, code for the functionality is written, and if it is written according to the expectations outlined in the unit test, the test will pass. Then, the programmer can refactor the code to make it efficient and easily understood, at which point the test should continue to pass.

In order to implement TDD into GRNsight, about 500 sample adjacency matrices were created to test for the various possible edge cases. These included tests for gene names, adjacency matrix modifications, matrix data, formatting, and special characters. Because the team’s main priority was to ensure that user provided data was parsed through correctly, all current unit tests exist on server side functions to ensure output data is accurate and a good representation of the adjacency matrix. Unit tests are executed using Mocha, a JavaScript test framework running on node.js, and are written using Chai, an assertion library for node.js.

A number of test functions were created that each check for proper functionality of specific errors and warnings in the GRNsight spreadsheet parser. In general, these test functions check for the frequency of a particular error or warning code, which is added to the errors or warnings array within the network object at detection of a mistake. Many test functions also gather additional information such as the exact location of any identifiable mistake, or the name of the affected gene. This additional information proves that the errors and warnings are caught correctly by the spreadsheet parser. Each of the 500 manually created are tested via 230 passing unit tests currently in the beta version of GRNsight (v2.0.1). Figure 6 shows that since implementing unit tests, coverage across all aspects of our code ranges from 68 to 85% (Table 1). Prior to our unit testing framework, each of these matrices were manually checked for functionality by uploading each sheet into GRNsight and ensuring that all aspects of the program cooperated with that sheet.



**Table 1.** Test coverage statistics in the master (v1.18.1) and beta (v2.0.1) version of GRNsight. There is consistent coverage across all sections of the code. Code coverage decreased between the two version because the project is in active development. Some test-last development (TLD) had to occur due to deadlines for our most recent publication.

The use of TDD consistently, and through all server-side functionality in GRNsight ensures that all of the errors and warnings mentioned earlier are caught and treated as expected. Without properly functioning error and warning handlers, the informative messages that go with them are of no use. Furthermore, TDD ensures that all functionality internal to the program is functioning as expected. Implementation of TDD to projects such as GRNsight is essential in ensuring the quality of the program and the corresponding visualizations.

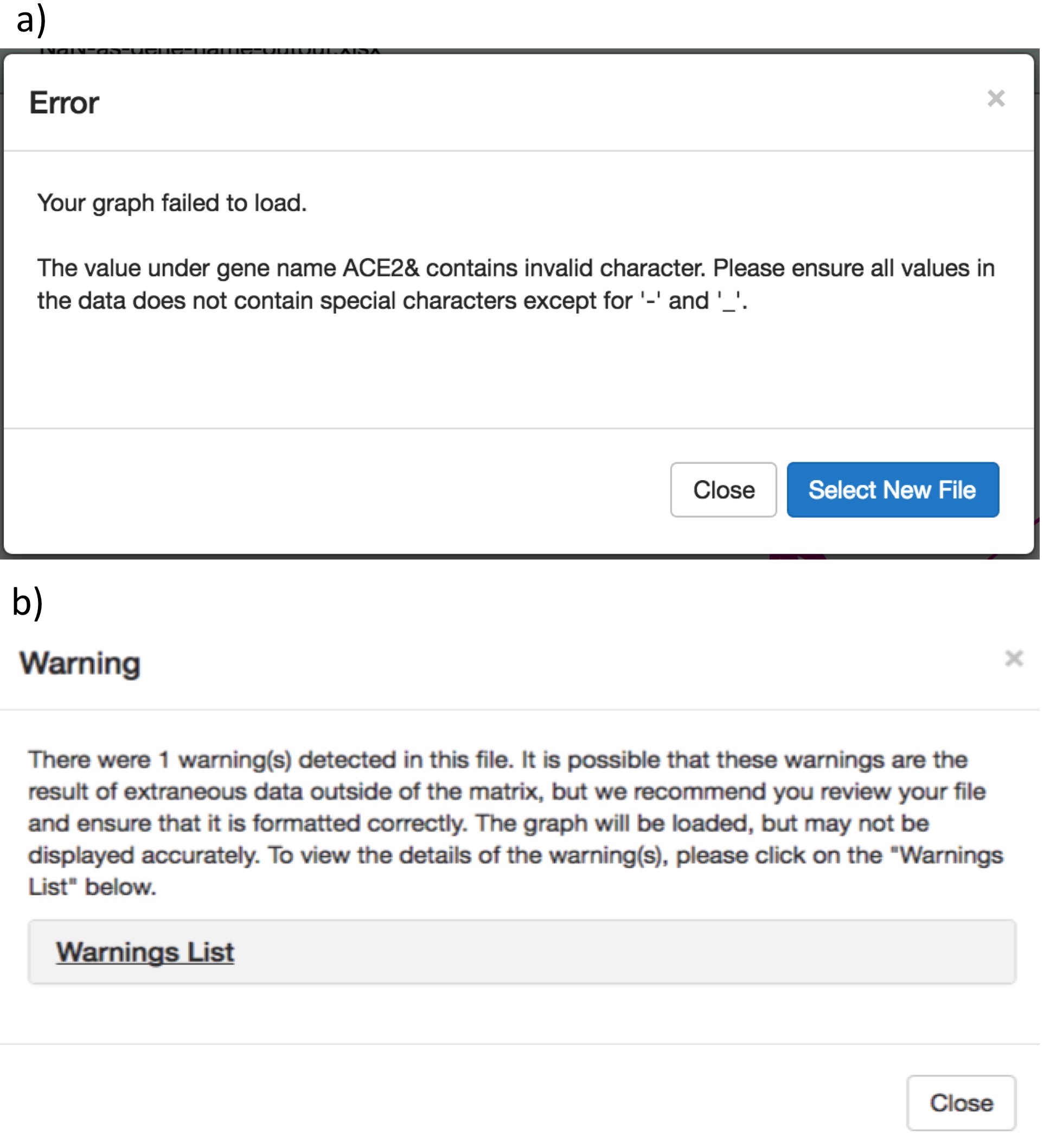
It was found in analysis of 27 different programs, made either in industry by professionals or in academia by students, that TDD improved both internal and external quality of the program (Rafique et al., 2013). Across a majority of these projects, the use of TDD also indicated a decrease in productivity. However, only one in nine academic projects reported decreased productivity upon implementation of TDD (Rafique et al., 2013). GRNsight is still in the process of optimizing the unit-testing framework, so it is difficult to say whether implementation of TDD has truly slowed down the productivity of the programmers. That said, because GRNsight is mainly used for analysis of biological data, the importance of accuracy in our visualizations outweighs the importance of quick implementation of functionality.

Whether or not TDD slows down productivity in the production of new functionality, it is essential to the program that all code functions as expected, and is checked for proper functioning immediately as it is being written. Test last development (TLD) offers too many opportunities for unit tests to conform to the preexisting code, and therefore does not ensure the same level of quality that TDD requires. While implementation of TDD has been a long process, it has been essential for the integrity of the visualizations produced by GRNsight, and conforms GRNsight to software engineering best practices.

II. Errors and Warnings Modal

In order to ensure that the data we present on GRNsight is accurate, two systems were put in place: errors and warnings catchers. The errors system was expanded so if a user attempts to upload an adjacency matrix that is improperly formatted or contains fatal issues that would compromise the accuracy of our visualization, GRNsight reject’s the adjacency matrix and returns an error modal that indicates the problem and a possible solution. While an error system was put in place prior to my onboard on the project, the errors that were checked for within the spreadsheet parser were limited. Warnings are a brand new addition to the GRNsight program. Non-fatal problems in the user provided adjacency matrix return an error to inform the user of the potential problem, but the graph is still presented because it is presumed that the error does not compromise the program overall.

Errors represent fatal mistakes in the adjacency matrix. These mistakes are fatal because they either cause our program or server to crash, or they force our visualizations to be inaccurate. In the case of an error, an error code is added to an errors array within a network object in the spreadsheet parser, and the entire network is returned without creating graph visualization. Instead, a three-part error modal (small “pop-up” window within the user interface) appears on screen to inform the user that there was a mistake in the adjacency matrix, what that mistake might be, and how the user could attempt to fix that mistake (Figure 5). Conversely, non-fatal problems in the user provided adjacency matrix return a warning to inform the user of the potential problem, but the graph is still presented because it is presumed that the mistake does not compromise the integrity of the program. In the case of a warning, GRNsight presents the user with a warning modal similar to the error modal that indicates that there is a non-fatal mistake in their adjacency matrix, what the mistake is, and how they can fix it. Users can click out of the warnings modal and view and analyze the graph, but have the opportunity to open it up once again with a link that exists underneath the Force Graph Parameters on the left-hand side of the screen. Because the mistakes that issue warnings are non-fatal to our spreadsheet parser, we allow the visualization to continue, but inform the user that there may be mistakes in the visualization due to these mistakes.



**Figure 5.** Error and warning modals. (a) The error modal features a three-part message to the user that explains what the problem is, why the problem exists, and how the user can approach solving the problem. It also features a button that allows users to easily upload a new file. (b) The warning modal features a similar three-part message as the error modal. It explains what the problem is, including why these problems are issued as a warning as opposed to an error. The collapsible “Warnings List” contains additional details about what the problem is, and how the user can approach finding the problem and fixing it.

To improve our errors and warnings checking system, we considered a number of edge cases, which represent a variety of mistakes a user might make when constructing their adjacency matrix prior to uploading it to GRNsight.

**Network Sizes**

Because GRNsight was optimized for small- to medium-scale networks, there are a maximum number of nodes and edges that we are able to accept. If a user attempts to upload a graph larger than 75 nodes or 150 edges, the program returns an error to let the user know that the graph is too large. The program limits the number of nodes and edges to this number because at this point, the visualizations on GRNsight become increasingly difficult to interpret as the number of nodes and edges increases, and the user would not be able to distinguish between the different edges and how they connect to one another. Furthermore, large networks cause our server to crash. This error ensures that GRNsight is being used with an ideal number of nodes and edges, and ensures that the visualizations produced by the program are accurate and readable.

While GRNsight will return an error at 75 nodes or 150 edges, it was optimized to work with no more than 35 nodes or 70 edges. If the user attempts to upload a network with larger than our recommended number of nodes and edges, an accurate graph visualization will still appear, but the program informs the user that the quality of the network graph may be compromised.

**Corrupt Gene Names**

Corrupt gene errors occur when users attempt to upload an adjacency matrix where the gene names are either too long or contain a special character other than a number, a dash (“-“), or an underscore (“\_”). Currently, our spreadsheet-parsing algorithm does not accept special characters, so uploading a spreadsheet with incorrect gene names would cause our program to crash. Additionally, long gene names cannot be presented within the nodes of our visualizations without stretching them to a point that they no longer fit the aesthetic requirements of our networks.

**Missing Gene Names**

Missing source genes and missing target genes issue a warning because, by default, our spreadsheet parsing algorithm assumes that any data underneath those missing gene names are irrelevant, and does not attempt to diagram them. Because this type of error is handled in our parser, and because all data provided by the user is presented accurately, we can issue a warning. This warning message is uniquely customized to include the exact cell that is missing data in the matrix to allow the user to quickly find the cell in their spreadsheet and act accordingly.

**Duplicate Gene Names**

If the same gene name exists within the source genes or the target genes (i.e. if there are two Abc1 genes listed as source genes) in the adjacency matrix, a duplicate gene error is triggered. If two genes of the same name are listed within the matrix, our program will be unable to distinguish between the two genes when creating the visualization. Because this would make our program crash, this is considered a fatal error.

**Data Type**

Microsoft Excel allows users to customize the data type of each cell in spreadsheets. Often, this causes the data within the cell to change format. For example, if the user has the value ‘0’ in a cell in the adjacency matrix, and changes the data type of that cell to “Time,” the value in the cell would change to ’00:00:00.’ Because GRNsight was created to only handle simple numerical relationships between genes, this format would cause our parser to crash, and therefore cause our program to crash. These types of fatal errors are caught as incorrect data type errors and are presented to users as such.

**Missing Data**

Finally, we created and error handler for the edge case where a user is missing data in an entire row in the adjacency matrix. We anticipate that this could be because a row of data is really missing within the sheet, or that the user has some stray data in their sheet that is being picked up by our spreadsheet parser. In the first case, we want to make sure that our user is aware of the missing data so they can either populate that row correctly, or delete that row altogether. In this case, the error in the sheet is truly an error and the user should correct their adjacency matrix to ensure that the visualizations are accurate, so the program returns an error to the user and the visualization is not presented.

In the second case, the user may have accidentally input data in a random cell in the sheet several rows away from their adjacency matrix. Because Excel spreadsheets contain upwards of 1048576 rows, random or accidentally typed data could potentially exist in a very large number of cells, and only one cell needs to carry extraneous data for GRNsight’s spreadsheet parser to catch it. For this reason, the GRNsight coding team implemented a strictness parameter that decides whether or not the extraneous data is a mistake or is intentional. Every row between the adjacency matrix and the extraneous data is considered a “missing row.” Therefore, if extraneous data exists 5 rows below the adjacency matrix, the spreadsheet parser would issue 5 missing row notifications to a counting function. Once the counting function indicates a number of notifications that exceeds the number in our strictness parameter, we change the type of notification we provide to the user.

Currently, the strictness parameter is set at 25. If there are fewer than 25 missing row notifications counted by our spreadsheet parser, we assume that the extraneous data was either written on purpose by the user, or is close enough to the adjacency matrix that the user could easily find it. Because of these conditions, if there are fewer than 25 missing row notifications, we inform the user that there is an error with their spreadsheet because of extraneous data. Conversely, if there are 25 or more missing row notifications, we assume that the extraneous data was added accidentally, or that it is too far away from the adjacency matrix for the user to find it easily. In this case, we issue a warning that tells the user where the extraneous data is, and continue to present the graph disregarding the extraneous data.

**Missing Adjacency Matrix Data**

Missing matrix data represented an instance where a user leaves a single cell within the adjacency matrix empty. By default, JavaScript assumes the empty cell to have a value of 0, and our spreadsheet parser acts accordingly. Because this type of mistake fits our requirements for a warning, we allow the graph to be presented on the website, and let the user know that there might be something wrong. Once again, we use a specialized error message that tells the user the exact cell with missing data, so adjustments, if necessary, can be made quickly.

These errors and warnings are essential to ensure that the program itself will not fail if a user uploads an improperly formatted adjacency matrix, and that the visualizations that GRNsight provides to users are accurate to the data provided. The informative three-part message in the errors and warnings modal allow users to fix any potential issues in their spreadsheet with ease, and continue working with the program. Prior to these additions to the GRNsight project, the spreadsheet parser only considered a few major edge cases, and did not have any warnings. Supplementing the previously created error list with additional error edge cases means that substantially more mistakes in adjacency matrices can be caught. Handling of these mistakes improves the integrity of the project overall, as the entire program is less likely to crash in the event that a user attempts to upload an incorrectly formatted adjacency matrix.

While error handling was meant to protect our program from crashing, the implementation of warnings ensured that the data we present to the user properly and accurately represents the biological information that the user uploaded. If a potential mistake was found in the adjacency matrix that did not cause the program to crash, we implemented a warning catch that returns a message to the user that informed them of the potential mistake. This ensures that the user knows the graph visualization may have problems, as well as what they should fix in order to completely prevent those problems from existing.

Both the error and warning handling systems are essential to communicating to the user when GRNsight does not function exactly as it is supposed to. However, these systems are only as useful as the messages they provide to the user. If the user doesn’t understand why GRNsight didn’t upload their adjacency matrix as they expected, the importance of the error and warning handling systems are diminished.

In order to convey this information to the user in the most optimal way, we used a three-part message system that tells the user: a) there was a problem when GRNsight tried to upload their adjacency matrix, b) where the problem was, and c) how the user can attempt to fix it. The purpose of the three-part error/warning message framework is to ensure that all error messages are precise and constructive, so users understand exactly where the problem is and how they can fix it. Furthermore, all messages were written so that lay-users could understand the entirety of the message. These characteristics are essential for a positive human-computer interaction as they ensure the user will be able to benefit from the message (Molich et al., 1990).

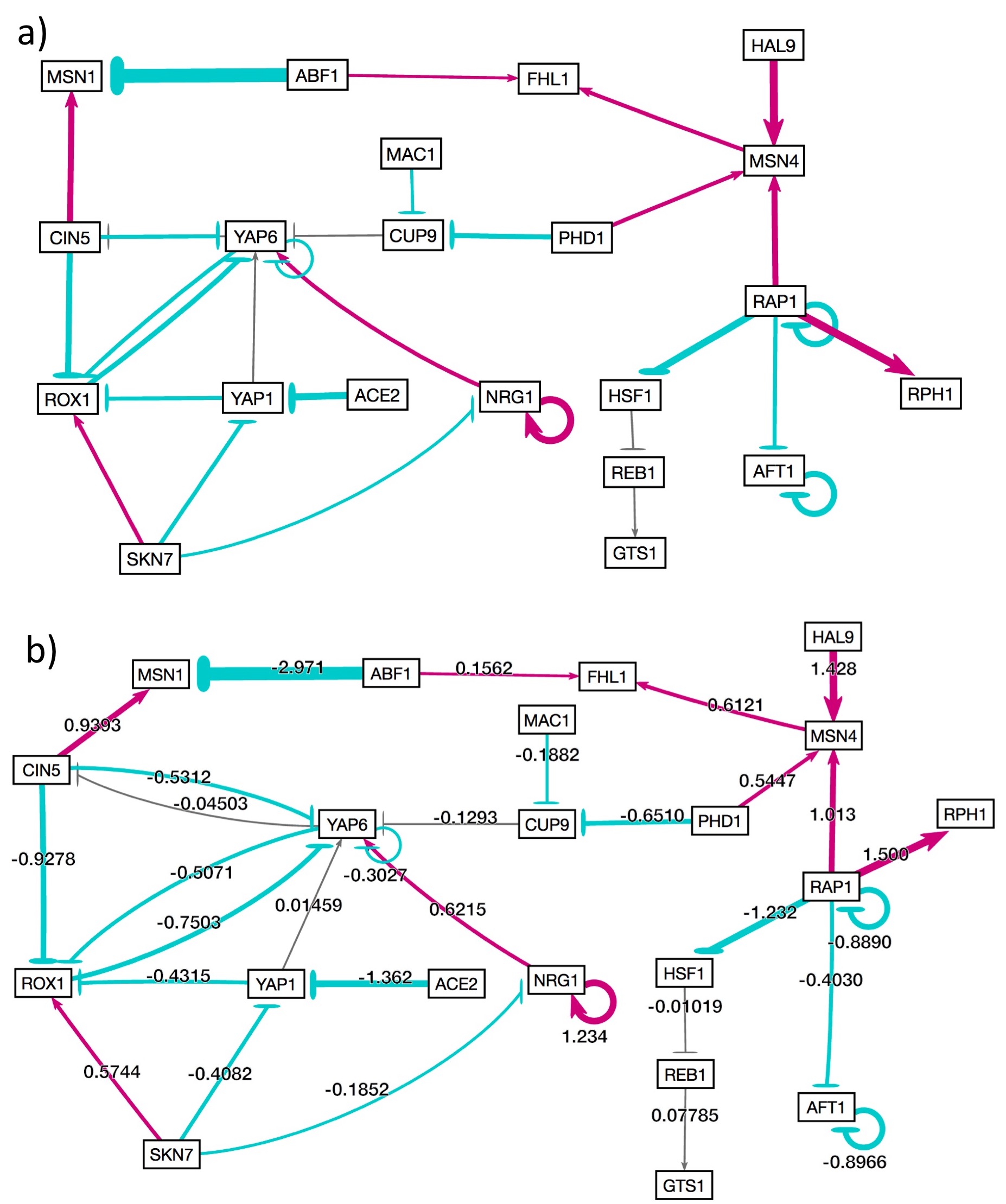
Moving forward in this aspect of the project, it would be beneficial to explicitly explain to the users how the potential problems in their adjacency matrix might affect their graph visualization in the event that the mistake only invokes a warning. For example, if a user uploads an adjacency matrix with a missing target gene name, it would be beneficial to inform the user that any regulatory relationships that influence the gene that belongs in the missing cell will not be presented in the visualization. This would not only make our warnings messages more comprehensive, but it will present all information in an easy to understand format for lay-users.

III. Front End Features

In addition to the errors, warnings, and unit tests, several front-end features were added to improve graph visualizations and allow users to further customize their network graphs. At GRNsights conception, it was decided that all matrix weight data would be normalized to fit within 12 distinct edge thicknesses. Each of these edges has an arrowhead proportional to the edge thickness. Initially, all arrowheads were configured to sit at the end of the edges with the same centering parameter. This became problematic, however, because arrowheads sat differently on each of the edge thicknesses. An important aesthetic feature that was adapted in order to combat this problem was giving each edge thickness and its corresponding arrowhead a different centering parameter. While this was supposed to fix the problem of centering arrowheads on edges, it was found that self-referential edges required their own centering parameter as well. After customizing each centering parameter according to the edge thickness and edge type, all arrowheads are properly centered.

Centering arrowheads to the edges was a purely aesthetic decision. Arrowheads were centered to each edge weight separately because using the same centering parameter across all edge weights resulted in off-center arrowheads at thicker edge weights. This adjustment was an important addition to the GRNsight graph visualization because one of the key aspects of GRNsight’s design is that graphs are quick and simple to produce, and are visually appealing.

Furthermore, a new feature was recently added to beta, v2.0.1, that allows users to always show edge weights via a floating number at the center of the edge, always high edge weights, or to only show edge weights upon mouse over via a small tooltip that appeared at the bottom right corner of the pointer on weighted graphs (Figure 7). Because unweighted graphs do not have specific edge weights, this functionality does not exist on unweighted graphs. This is especially useful for researchers who want a more complete picture of their network of interest, or want to use our networks in a presentation.



**Figure 7.** Showing and hiding weights. When an adjacency matrix is uploaded to GRNsight, default functionality is to only show weights upon mouse over in a tooltip that appears on the bottom right corner of the pointer. The user can select to always have weights hidden using menu options that exist in the left side bar, and under the Format tab in the file menu. In either case, the network graph would appear as it does in (a). If the user chooses to always show weights, the data would appear in the center of each edge as show in (b).

When presenting and analyzing networks, it is often beneficial to see both graphic displays of the regulatory relationships (i.e. via color coded edges with varying thicknesses) as well as numerical weights. Allowing users to always show weights means they can easily interpret their graphs without having to mouse over each individual edge. Additionally, hiding all weights can be beneficial if the user chooses to not have extra data on their network.

Another important reason for users to be able to always show weights is because it allows users to compare networks fairly. If two networks are generated using GRNsight, in order to compare the graphs with each other, the user must be able to look at the numerical values of each edge to understand how the different regulatory relationships differ from each other. A new feature that is currently under implementation is customization of the normalization factor. While users can currently compare numerical weight data using the “always show weights” setting, this does not modify the thickness of the edges in the network itself.

Allowing users to adjust the normalization factor means users can select a range of edge weights encompassing both networks they wish to compare, and edge thicknesses will be decided according to that range of edge weights. If a user inputs the same normalization factors into our user interface, the edge thicknesses will automatically adjust according to that factor. If multiple graphs are adjusted to the same normalization factors, the edge thicknesses of those graphs will become entirely comparable. Customization of normalization factors will be achieved using both minimum and maximum edge weights from all comparable networks. The range of edge weights will be compartmentalized into 12 edge thicknesses using D3.js, and the resulting visualization will use edge weights according to the newly compartmentalized weight-to-edge thickness settings.

Error and warning catching, TDD, and the other user interface features mentioned above are necessary and indispensable additions to the GRNsight project. These additions not only conform to software engineering best practices, but also attempt to improve user experience overall. The expanded error and warning handling system encompasses several new edge cases, and ensures that GRNsight runs correctly, and that visualizations on GRNsight are created accurately. TDD encourages student programmers on the project to code new unit test prior to coding new functionality, which promotes internal and external quality of GRNsight code. Finally, adjustments to the visualization via aesthetic modifications and customization of the appearance of edge weights mean users can create network graphs for various different uses, and even compare several networks more easily. Moving forward, customizable normalization factors will continue to improve GRNsight, and will be a new feature in v2.0.1.

**References:**

Bissi, W., Neto, A. G. S. S., & Emer, M. C. F. P. (2016). The effects of test driven development on internal quality, external quality and productivity: A systematic review. *Information and Software Technology*, *74*, 45-54. doi: 10.1016/j.infsof.2016.02.004

Bastian, M., Heymann, S., & Jacomy, M. (2009). Gephi: an open source software for exploring and manipulating networks. *International AAAI Conference on Web and Social Media*. *8*, 361-362. Retrieved from http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154

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

Kitano, H. (2002). Systems biology: a brief overview. *Science*, *295*(5560), 1662-1664. doi: 10.1126/science.1069492

MacNeil, L. T., & Walhout, A. J. (2011). Gene regulatory networks and the role of robustness and stochasticity in the control of gene expression. *Genome research*, *21*(5), 645-657. doi: 10.1101/gr.097378.109

Pastrello, C., Otasek, D., Fortney, K., Agapito, G., Cannataro, M., Shirdel, E., & Jurisica, I. (2013). Visual data mining of biological networks: one size does not fit all. *PLoS Comput Biol*, *9*(1), e1002833. doi: 10.1371/journal.pcbi.1002833

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

Wiese, R., Eiglsperger, M., & Kaufmann, M. (2004). yfiles—visualization and automatic layout of graphs. In *Graph Drawing Software* (pp. 173-191). Springer Berlin Heidelberg. doi: 10.1007/978-3-642-18638-7\_8