Anindita Varshneya

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

Materials and Methods/ Results

**Results:**

Errors and Warnings Modal

In order to ensure that the data we present on GRNsight is accurate, two systems were put in place. First, we expanded our errors system so that 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, we reject the adjacency matrix and return 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.

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 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, warnings are non-fatal mistakes in the adjacency matrix. They are mistakes that do not cause any fatal issues to our parser, and do not compromise the accuracy of our visualizations given the data the user uploaded to the website. In this case, we present the visualization, but also immediately present a warnings modal with the same three-parts as mentioned for the error modal (Figure 5). In this case, the user has the option to close out of the warnings modal and continue working with their visualization. If, at any point, the user would like to reopen the warnings modal, they can do so using a hyperlink underneath the Force Graph Parameters on the left hand side of the screen.

To improve our error 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. The types of errors we check spreadsheets for include: large network size, corrupt genes, duplicate gene names, incorrect data types, and empty rows. 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. We limit the number of nodes and edges to this number because at this point, our visualizations are incredibly difficult to interpret, and the user would not be able to distinguish between the different edges and how they connect to one another. Furthermore, incredibly 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.

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.

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.

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.

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. In this case it makes sense to return an error to the user and prevent the visualization from being created.

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 our spreadsheet parser to catch it. For this reason, we implemented a strictness parameter that decides whether or not the extraneous data is a mistake or is on purpose. Every row between the adjacency matrix and the extraneous data is considered a “missing row.” Therefore, if there was extraneous data 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, our 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.

In fact, in the case of any of our warning conditions, we present the user with a warning modal that indicates that there is a non-fatal mistake in their adjacency matrix, what the mistake is, and how they can fix it, and also present the user with the visualization. Users can click out of the warnings modal and view and analyze the graph, and can 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. The conditions in which we issue a warning include: empty rows (as mentioned above), missing source genes, missing target genes, missing matrix data, and network size warnings.

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.

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.

Network size warnings mimic the network size error mentioned earlier. While GRNsight will return an error at 75 nodes or 150 edges, it was optimized to work best 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, the graph will still appear, but we inform the user that the quality of the network graph may be compromised, while it is still accurate.

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.

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 being caught accurately, we implemented Test Driven Development (TDD) to GRNsight. TDD is a software development best practice that promotes the developer to make incremental changes to their program, and ensure that it is running expectedly after each incremental change using unit tests (Bissi, 2016). More explicitly, a unit test ensures proper behavior for a particular unity 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 write the code for that functionality, and if they wrote it according to their predetermined expectations as made explicit in the unit test, the unit test should pass. At this point, the developer can go back and refactor their code to ensure it is efficient and easily understandable, and confirm that no matter what changes they make to their code, their unit tests always pass. This process is especially important for academic research projects, such as GRNsight, because this ensures internal software quality of our project (Bissi, 2016).

In order to implement TDD into GRNsight, about 500 sample adjacency matrices were created to test for the various possible edge cases, as mentioned in the previous section. These included tests for gene names, adjacency matrix modifications, matrix data, formatting, and special characters. Because our 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.

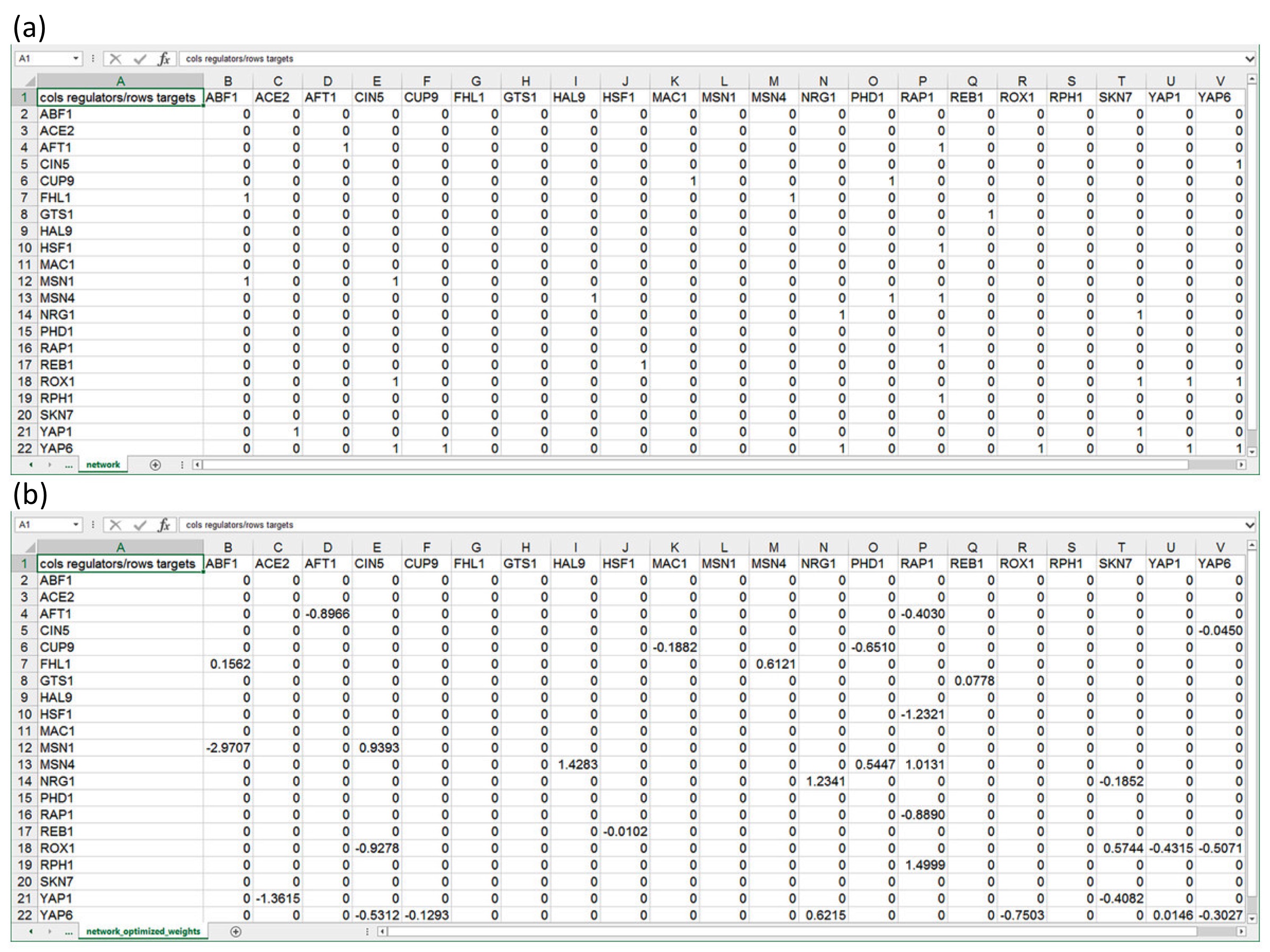
We created a number of test functions that each checked for proper functionality of specific errors and warnings in the GRNsight spreadsheet parser. In general, these test functions check for the frequency of a particular error or warning code, which is added to the errors or warnings array within the 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 by are parser, and are caught correctly, and ensures that the error and warning messages presented to the user are accurate as well. 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.

Front End Features

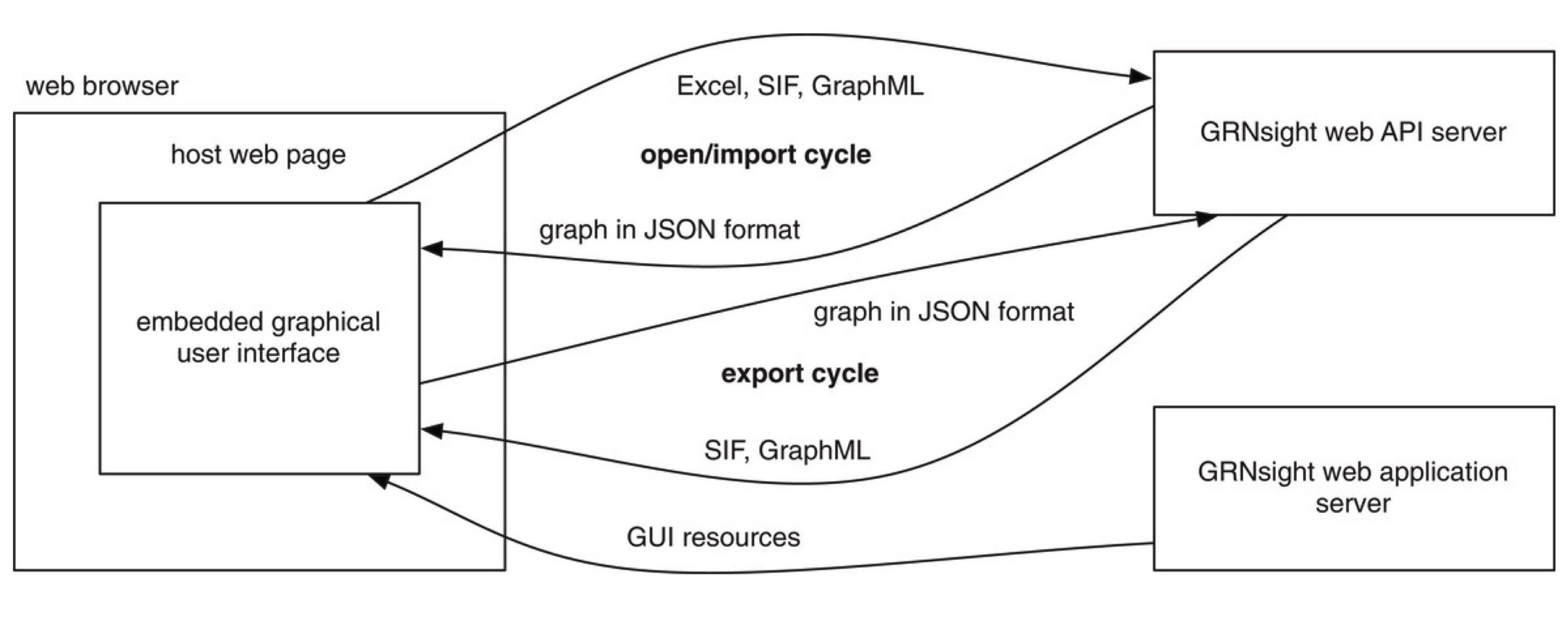
In addition to the errors, warnings, and unit tests, several front-end features were added to improve graph visualizations and allow users to further customize their network graphs. At GRNsights conception, it was decided that all matrix weight data would be normalized to fit within 12 distinct edge 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.

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.

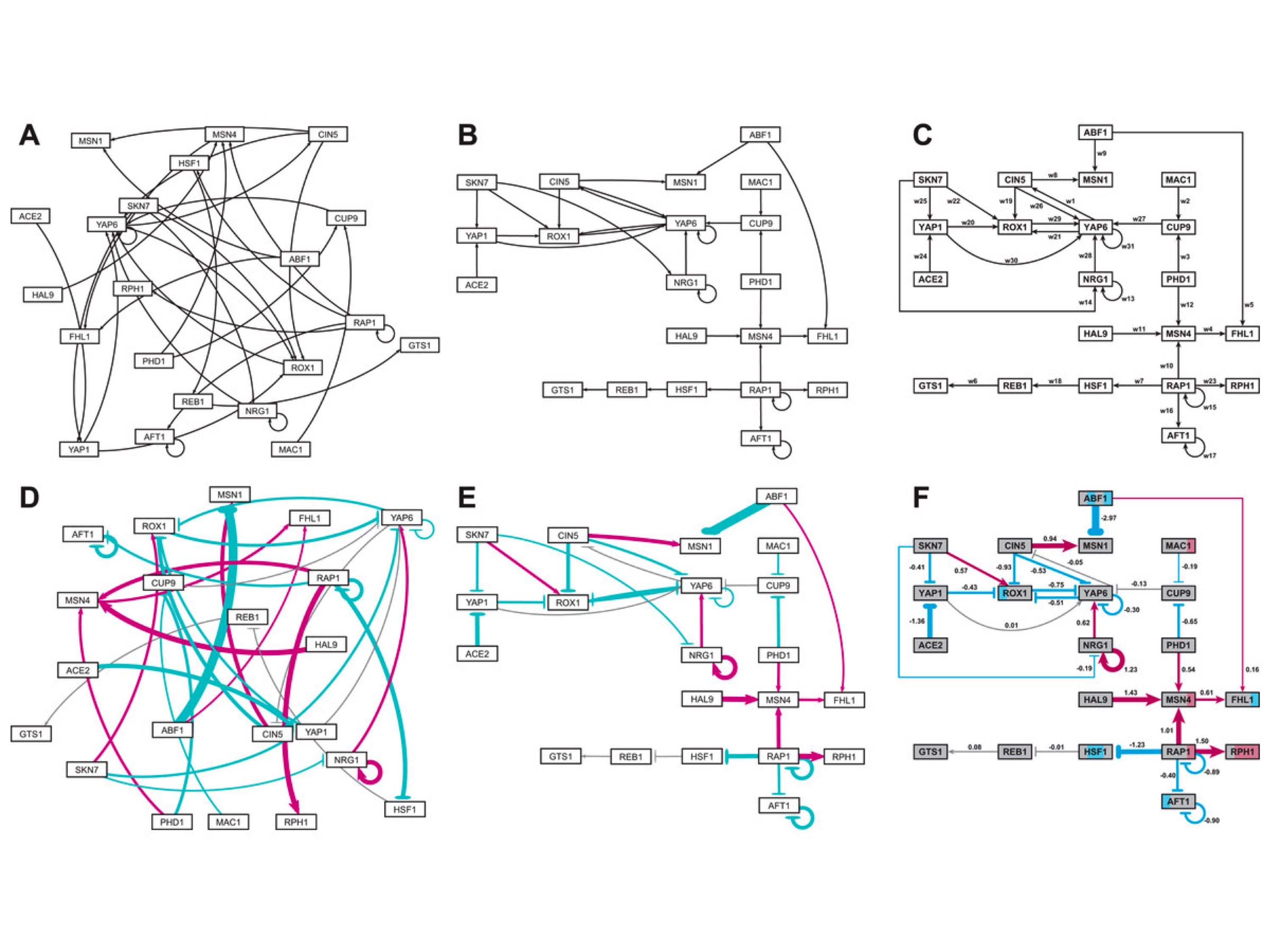
**Figures:**



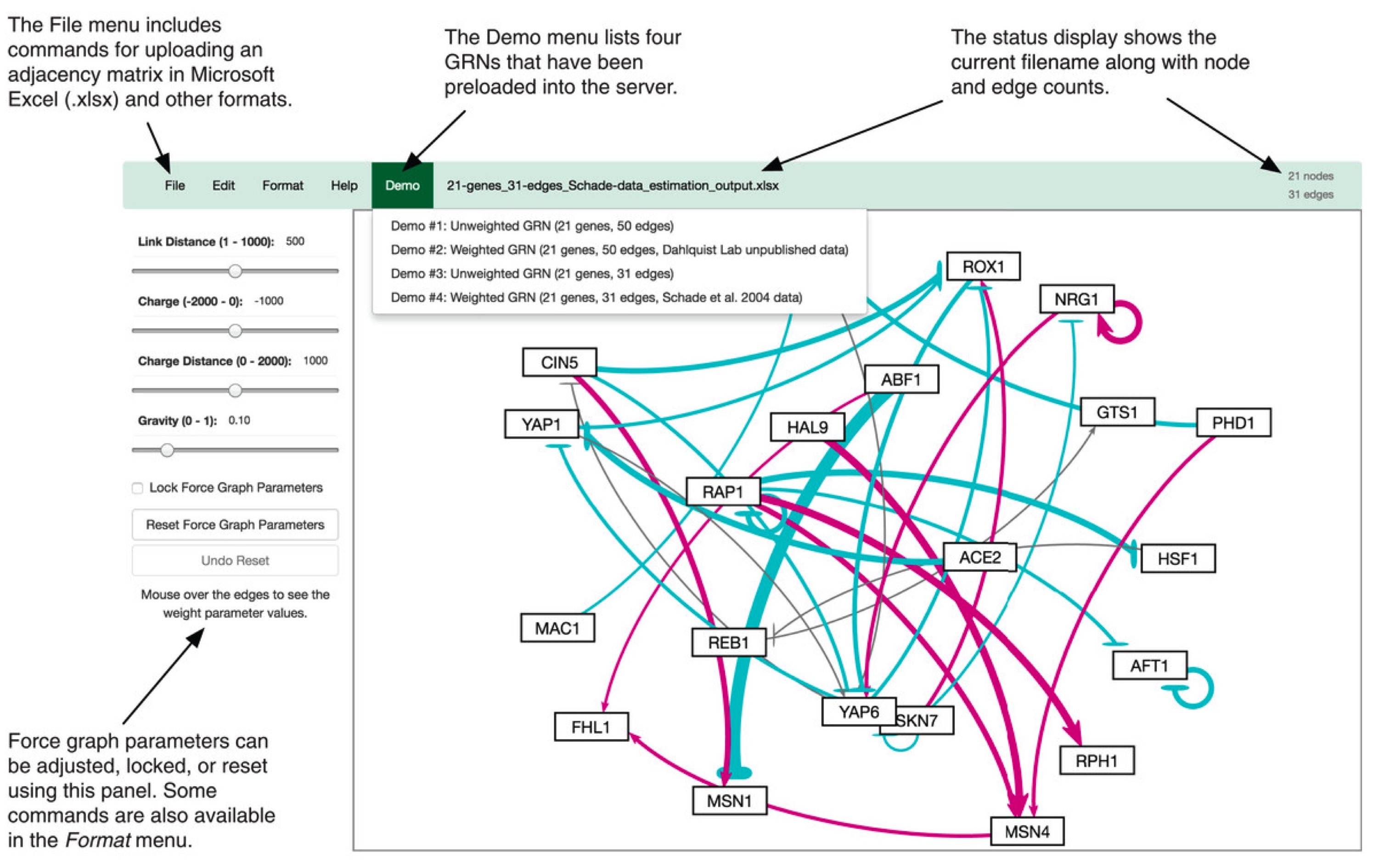
**Figure 1.** Properly formatted adjacency matrices that can be opened in GRNsight to construct network visualizations. All matrices are directed. Gene names in column headings represent regulators, and genes in row headings represent target genes. (a) An unweighted adjacency matrix where ‘0’ represents 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) [this will be mentioned in the introduction]



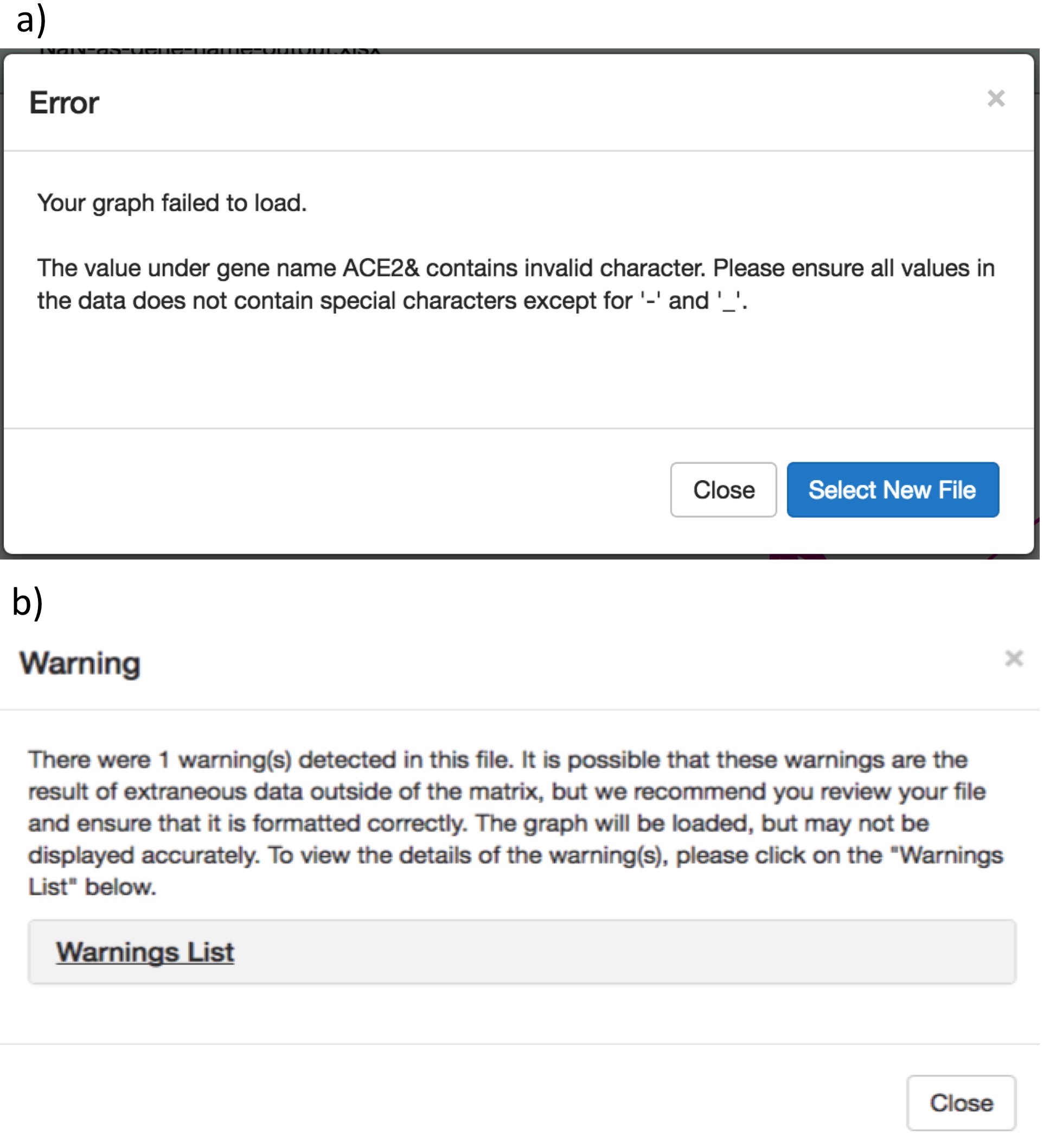
**Figure 2.** GRNsight has a complex, server-oriented architecture. The web browser holds the web page and the embedded graphical user interface. When a network spreadsheet is opened on GRNsight, the GRNsight web API server parses the spreadsheet and sends it back to the web browser in JSON format, where 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) [this will be mentioned in the introduction section]



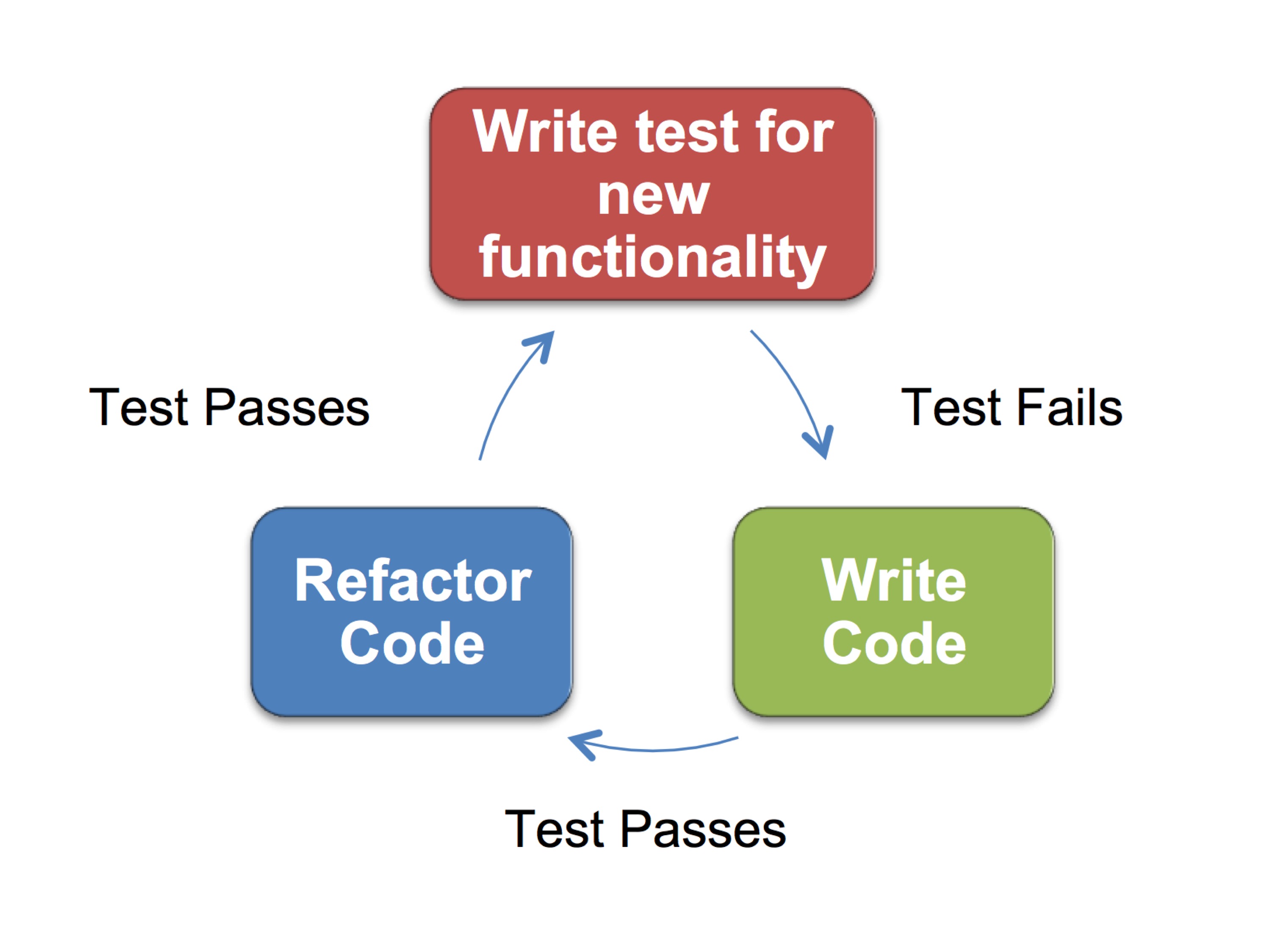
**Figure 3.** Medium-scale gene regulatory networks. (a) and (d) are unweighted and weighted gene regulatory networks, respectively, automatically produced on GRNsight within about 10 milliseconds. In the weighted network, magenta edges represent activating relationships, while cyan edges indicate repressing 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) [this will be mentioned in the Introduction]



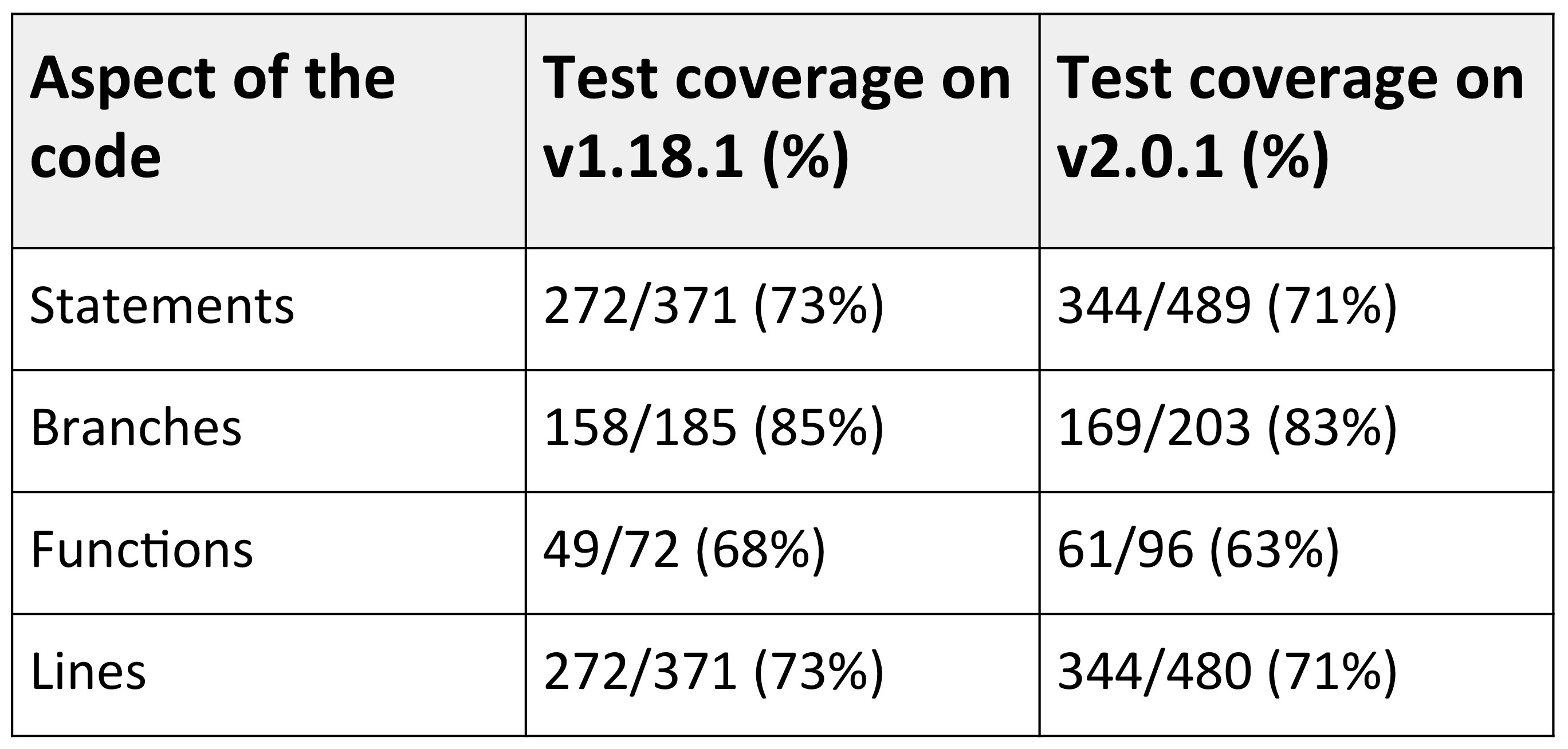
**Figure 4.** An annotated screenshot of the GRNsight user interface. The file menu at the top left of the screen provides commands for uploading an adjacency matrix in Microsoft Excel (.xlsx) and other formats. The demo menu provides four sample adjacency matrices preuploaded on the server. The status 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) [this will be mentioned in the introduction]



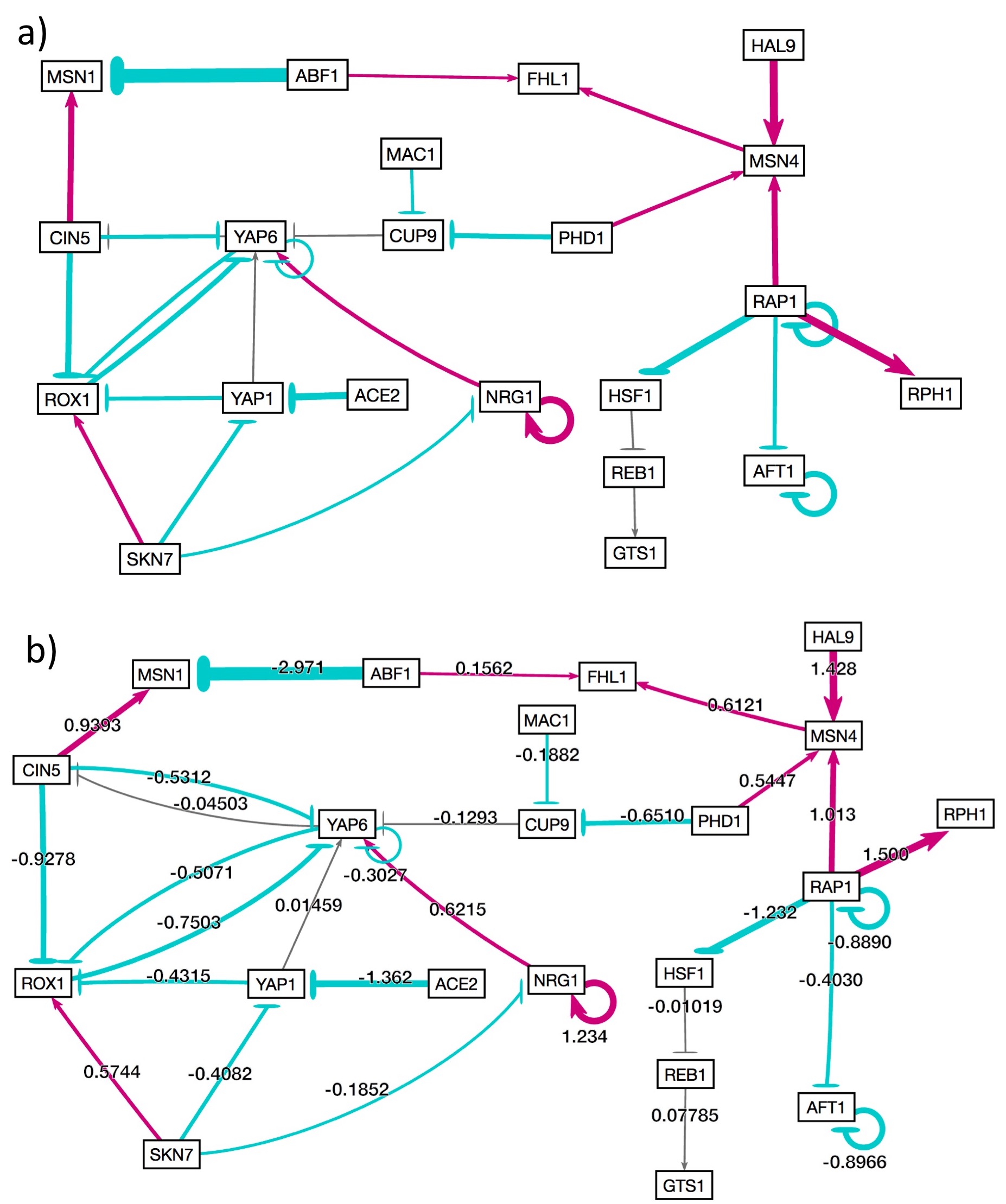
**Figure 5.** Error and warning modals. (a) The error modal features a three-part message to the user that explains what the problem is, why the problem exists, and how the user can approach solving the problem. It also features a button that allows users to easily upload a new file. (b) The warning 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.



**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.



**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.



**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).

**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

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