Linked, Interactive Views of Linked Data

Elizabeth Whalen

August 20, 2006

1 Overview

1.1 Goals and Definitions

The *iSNetwork* package lets users perform exploratory data analysis by creating linked, interactive views of linked data sets. In addition to providing the functionality to create linked, interactive views of linked data, one of the goals of this package was to create a design that is extensible so that users can make additions based on the needs of their data. These goals are discussed in more detail in the following paragraphs.

First definitions of linked views, interactive views and linked data sets are given in the following paragraphs. Linked views mean that if a component that represents data (such as a point on a plot or a row in a spreadsheet) changes its appearance on one view, then the corresponding component on a second view also changes its appearance. For components on different views to be corresponding, the data displayed in those components must come from the same entity in a model. As an example, suppose that the model consists of the gene expression levels from four samples. If the same one hundred genes are studied in the four samples, then the model could be represented as a matrix of four columns by one hundred rows with four columns for the four samples and one hundred rows for the one hundred genes. In this model, the entities are the genes. If two scatter plots are created as views of this model, where the first plot displays sample one versus sample two and the second plot displays sample three versus sample four, then the points on each plot that referred to the same gene are corresponding components. Linked views, which have a long history in statistics, are now considered a standard feature of interactive data visualization software Swayne et al. (2003). Linked views have been implemented in several pieces of statistical software, such as XLisp-Stat Tierney (1990) and GGobi Swayne et al. (2002). Implementing linked views is based on the model-view-controller (MVC) design Gamma et al. (1995), which is a widely used and well understood paradigm.

Having interactive views means that when the user interacts with a view, a response occurs. As an example of interactivity, suppose that when the user clicked on a point in a view, the point was colored red. The interactivity is richer and more flexible if multiple events can be noticed (such as a mouse click, a mouse movement, or a key press event) and if the response to those

events can be modified. To provide this interactivity, the *iSNetwork* package requires the *RGtk* and *gtkDevice* packages, which let the users interact with Gtk functions from the R interface. Gtk is an open-source X Window toolkit for creating user interfaces. More information about these packages is given in Section 1.2.

Linked data sets mean that the data have more than one conceptual grouping, but there is a relationship between these different conceptual groupings. Linked data sets are an idea that is familiar to users of relational databases. Data in a relational database are typically stored in many linked tables with rows in different tables linked to each other through a unique identifier called a key. In this example, each table is considered a data set and they are linked through the keys. Another situation where linked data sets occur is when there are experimental or study data that are linked to meta-data. An example of this situation is microarray data, as the experimental data, that is linked to meta-data, such as the Gene Ontology (GO) graph that gives the molecular functions of the genes studied in the microarray experiment. In the *iSNetwork* package, we are most interested in experimental data that is linked to meta-data so the linking is modeled as a parent-child relationship. Thus, in the example given here, the microarray data is the parent data set and the GO graph is the child data set because the GO graph is determined by the genes that are in the microarray data.

Creating linked, interactive views of data is flexible for exploring multivariate data because many different views can be created, such as scatter plots, spreadsheets, and heatmaps, and these views are linked because they are based on the same underlying data. Thus, users can decide which views best visually represent their data. Also, since the views are interactive, users can change the views while looking at them to make the views more informative about the underlying structure in the data. This flexibility in linked, interactive views gives users a powerful tool for visually exploring their data. However, linked views can not solve all problems. Clearly, only a few linked views can be viewed simultaneously by the user because only a certain number of views can be shown on a computer screen. Thus, there is a limit to the number of dimensions that can be represented by linked views.

Finally, an extensible design is imperative so that future users can make additions based on their needs. The design for creating linked, interactive views is based on the model-view-controller paradigm. The *MVCClass* package, which is required by the *iSNetwork* package, defines classes and generic functions that implement the model-view-controller (MVC) paradigm. The MVC paradigm is a design that consists of three types of objects: the controller, which defines what actions occur in response to user input; the view, which consists of displays of the data; and the model, which manages the data. The power of the MVC design is that it decouples the views of the model from the model by creating a subscribe/notify procedure between them. In other words, the views subscribe to a particular model and the model must notify the views when a change occurs so that the views are updated. By separating the model from its views, only one copy of the model needs to be stored. Please see the vignette for the *MVCClass* package for more information about the classes that are defined in that package. The use of these classes allows an extensible design because the inheritance structure of the classes lets users create new model and view classes that inherit from already implemented classes.

Besides using the MVC paradigm, several other aspects of the *iSNetwork* have been carefully

designed for extensibility. These include letting the user add menus to the graphical user interface (GUI), letting the user change the response to an interaction with a view, and letting the user create new methods for generating and linking data sets. A full discussion of extensibility in the *iSNetwork* package is given in Section 14.

One important thing to notice about the extensibility of the *iSNetwork* package is that any additions the user may want to make can all be performed in the R language. If the user wants to define new model or view classes, this is done in R and similarly if the user wants to add new menus, this also is done in the R language. Thus, even though the *iSNetwork* package requires other packages that use other languages, such as C, making additions to the *iSNetwork* package only requires that the user can program in R. More information about the extensibility of the *iSNetwork* package can be found in Section 14.

1.2 Required Packages

The required packages for *iSNetwork* are *graph*, *Rgraphviz*, *RGtk*, *gtkDevice*, *MVCClass*, *BioMVCClass*, and *Biobase*. A short description of why these packages are needed by *iSNetwork* is given here. One of the data structures that users can study is a graph and thus, the *graph* package is required to create instances of a graph and the *Rgraphviz* package is necessary to create views of these graph ojects. Another data structure that users may want to study is the `ExpressionSet` and thus, the *Biobase* package is required. The *RGtk* package is used to create a GUI through which the user can load, view, plot and interact with the data. The *gtkDevice* package creates a device of type Gtk that looks like a X11 device, but a Gtk device can respond to events, such as a button press or a mouse over event. By generating a function call in response to an event, interactive views can be created. Finally, the *MVCClass* and the *BioMVCClass* packages define the classes and generic functions that are used in the *iSNetwork* package.

1.3 GUI versus Command Line Functions

The functionality that is implemented in the *iSNetwork* package can be accessed either through the GUI or through command line functions. Both methods for accessing the functionality are explained in this vignette.

2 Getting Started

```
> library(iSNetwork)
```

After loading the library, the *iSNetwork* functionality can be accessed through either the command line functions or the GUI. To learn how to open the GUI, continue with Section 2.1.

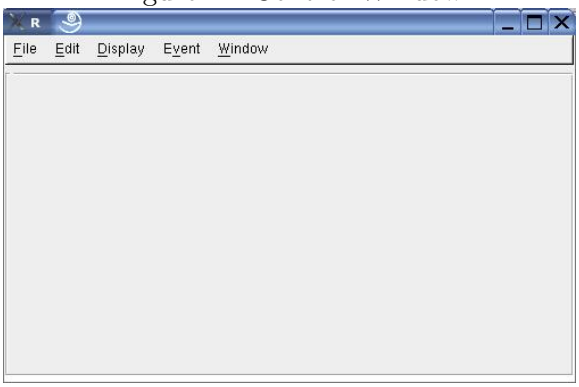
2.1 Opening the GUI

After loading the library, the first command is to open the control window, using createControlWindow, if the user intends to use the GUI. Data can be loaded, viewed and plotted by selecting menu items on the control window.

```
> createControlWindow()
```

The control window looks as shown in Figure 1.

Figure 1: Control Window



3 Loading Data

Currently, this package can load and create views of a data frame (or a matrix), an ExpressionSet, a graph, or a gene set enrichment model. For information on the gene set enrichment model please see the *BioMVCClass* Vignette and for information on gene set enrichment analysis, please see Subramanian et al. (2005). Support for other data classes can be added later.

Before creating views of the data, it first must be loaded through the GUI or through the command line functions. The user can continue to load as many data sets as needed through the GUI or through the command line functions, but only one data set is considered the active data set. This concept of an active `MVC` is discussed in Section 4.

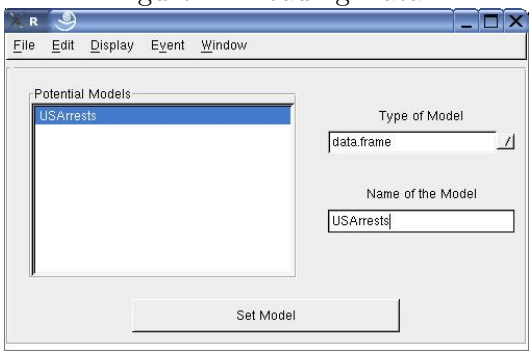
When a data set is loaded, an object of class `MVC` is created that stores the data set and all views of that data. The `MVC` class is defined in the *MVCClass* package and it is used to tie the model, view, and controller objects together into one object that revolves around one model (one loaded data set). Since there is a one-to-one relationship between the model and the `MVC` object, they both are referred to by the same name and when the active data set is discussed, this also refers to the active `MVC` object.

3.1 Loading Data Through the GUI

To load data through the GUI choose the Load Model menu item under the File menu or alternatively, press Ctrl-L to activate the Load Model menu item. Users can choose which type of data they want to load by setting the 'Type of Model' drop down box. The three options for type of model are `data.frame`, `ExpressionSet`, and `graph`. Notice that the gene set enrichment model is not available through this interface. At this time the gene set enrichment model can only be loaded by creating a child model, which is discussed in Section 5. Once the user has chosen the type of model, the names of all data sets of that class that are loaded in R's global environment are shown in the 'Potential Models' list. Thus, it is expected that any data to be loaded are already available in R's global environment. Next highlight the name of a data set to load from the 'Potential Models' list. Then type in the name of the model in the text box (this can be any name that has not already been used for a loaded model) and click on the 'Set Model' button. Now the new model is loaded.

To make the control window look as it is shown in Figure 2, complete the following steps. First load the USArrests data set into R's global environment by typing data(USArrests) at the R prompt. Then choose the Load Model menu item under the File menu. Set 'Type of Model' to `data.frame`, highlight USArrests in the 'Potential Models' list, and type in the name of the model.

Figure 2: Loading Data



The last step is to click the 'Set Model' button. Now the USArrests data set has been loaded and interactive views of the data can be created. The frame around the control window now contains the text, USArrests, which shows that the active data set (and active `MVC` object) is the USArrests data set.

3.2 Loading Data Through the Command Line

The user can load any data that are of class `data.frame` (or `matrix`), `graph`, or `ExpressionSet`. The loadData function loads data that are already available in R's global environment. Note that loading data means that a `MVC` object is instantiated that contains the data, which then lets the user create linked, interactive views of that data. To load data into a `MVC` object, the

data must initially be loaded in R's global environment. The example code below loads a data frame data set, an ExpressionSet data set and a graph data set. Please see the man page for the loadData function for more information.

```
> loadData(USArrests, "USArrests", "data.frame")
> library(CLL)
> data(sCLLex)
> loadData(sCLLex, "CLL", "ExpressionSet")
> set.seed(123)
> V <- letters[1:10]
> M <- 1:4
> g1 <- randomGraph(V, M, 0.2)
> loadData(g1, "testGraph", "graph")
```

3.3 Loading Model Variables

After a model has been loaded, it is possible to add extra variables to the model by loading model variables. As an example, suppose that the user has loaded `ExpressionSet` data and now wants to add row t-test statistics for each gene to the model. This can be done by adding a model variable to the model. Model variables are extra information about the model data. Extra data that the user may want to show in views of the model should be stored as a model variable. Continuing the previous example, the user may want to show the t-test statistics as tooltips in the views.

In our example in Sections 3.3.1 and 3.3.2, we load model variables that are booleans that indicate whether genes in the CLL model have passed a filter. These data are stored in the CLL package. The nsFilter boolean variable tells whether the genes in the `sCLLex ExpressionSet` passed the nonspecific filter and the sFiltertBH boolean variable indicates whether the genes in the `sCLLex ExpressionSet` passed the specific filter. Please see the man page for these variables for more information.

```
> data(nsFilter)
> data(sFiltertBH)
```

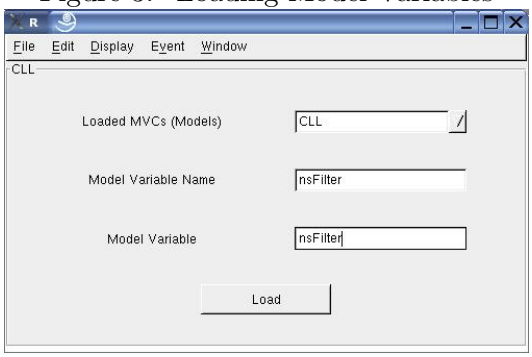
3.3.1 Loading Model Variables Through the GUI

To load a model variable through the GUI, choose the Load Model Variables menu item under the File menu or alternatively, press Ctrl-A to activate the Load Model Variables menu item. Then choose which loaded MVC (model) this model variable will be attached to. In our example, we will attach our variables to the CLL model that was loaded in Section 3.2. Next give the name of the new model variable by typing in the Model Variable Name text box. This name is used to refer to the model variable whenever you want to use it. Finally fill in the

Model Variable text box. This is the name of the variable in R's global environment. Note that these two text boxes do not have to be the same, but they can be.

In our example, where we are adding two boolean variables to the CLL model, we start by adding the nonspecific filter boolean variable. The values put in the text boxes are shown in Figure 3. The model variable name is 'nsFilter' and the model variable is 'nsFilter'.

Figure 3: Loading Model Variables



Finish by clicking the Load button. A new window will appear that says 'nsFilter was loaded as a model variable.'

Next add the specific filter boolean variable to the CLL model by giving a model variable name of 'sFiltertBH' and a model variable of 'sFiltertBH'. Again click the Load button and a window appears that says 'sFiltertBH was loaded as a model variable.'

3.3.2 Loading Model Variables Through the Command Line

To load model variables through the command line, you use the loadModelVar function. In the code below, the nonspecific filter and specific filter boolean variables are added as model variables to the CLL model.

```
> loadModelVar("CLL", "nsFilter", nsFilter)
> loadModelVar("CLL", "sFiltertBH", sFiltertBH)
```

4 Setting the Active MVC

As mentioned in Section 3, whenever a data set is loaded a `MVC` object is created that binds the data and its views into one unit. There is a one-to-one relationship between the `MVC` object and its data set so they are referred to by the same name. In the *iSNetwork* package, the `MVC` object is considered the unit we work with. Though multiple `MVC` objects can be created (because multiple data sets can be loaded), only one `MVC` object can be active at a time.

The first MVC object that is created when the first data set is loaded automatically becomes the active MVC object. Thus, if the user initially loaded the USArrests data set, then the MVC object called USArrests is the active MVC object. Any other data sets loaded after this first data set are not the active MVC object. The user can tell what the active MVC object is by looking at the name shown in the frame around the control window, underneath the main menu, or by calling the function, getActiveMVC.

Being the active MVC object means that all of the menu items or the command line functions refer to operations performed on this MVC object. Thus, if the user loads a second data set after loading USArrests, USArrests is the active MVC object and the user is able to create views of the USArrests data set. If the user instead wants to create views of the second data set, then the user has to set the second MVC to be the active MVC. Setting the active MVC can be performed through the Set Active MVC menu item under the Edit menu or by calling the function, setActiveMVC.

Once the interesting MVC object is the active MVC, the user can create views of the data. Note that different types of views can be created depending on the class of the data. For example, if the data are of class data.frame, then spreadsheets and scatter plots can be created. Creating views of data.frame objects are discussed in Section 7, views of graph objects are discussed in Section 8, and views of ExpressionSet objects are discussed in Section 9.

4.1 Setting the Active MVC Through the GUI

Suppose that the user has loaded three data sets: USArrests, which is of class data.frame, testGraph, which is of class graph, and CLL, which is of class ExpressionSet. These are the three data sets that have been loaded so far in this vignette using the command line function, loadData. Now to set testGraph as the active MVC through the GUI, highlight the Edit menu and select the Set Active MVC menu item. Similarly, the user can press Ctrl-V to activate the Set Active MVC menu item. A new window appears with radio buttons to allow the user to choose which of the MVC objects is the active MVC. This new window appears as shown in Figure 4.

Figure 4: Set the Active MVC

Now if testGraph is the MVC object that the user is interested in, choose the testGraph radio button and click the Set button. Now the active MVC is testGraph.

4.2 Setting the Active MVC Through the Command Line

To set the active MVC object through the command line, call the function, setActiveMVC, with the name of the MVC object, which will become active. The example code below sets the CLL MVC as the active MVC. Please see the man page for this function for more information.

To see the names of all loaded MVC objects, call the function, getModelNames.

```
> getModelNames()
> setActiveMVC("CLL")
```

5 Creating a Child MVC Object

Recall from Section 3 that when a data set is loaded, a MVC object is created that holds the data and any views of the data. Once a MVC object has been created by loading data, the user can create a new MVC object that is a child of the loaded MVC object. To create a child MVC, a function is called that creates a new model (and thus, a new MVC object) by performing some type of operation on the original MVC's model data. Thus, users can create a new MVC object in two ways: they can load data stored in R's global environment, which was discussed in Section 3, or they can create a new model that is derived from an existing MVC's model data, which is discussed in this section. As a simple example, consider that the original model data is a data frame and the function that creates a child MVC creates a subset of the data frame. This subset of the data frame is the model data for the new child MVC. These two data sets are clearly linked since the child data set is just a subset of the parent data set. With the simple example given here, the parent MVC contains the full data frame and the child MVC contains a subset of the data frame. More examples of creating child MVC objects are given throughout this section.

Clearly, the functions to create a child model depend on the class of the original model. For example, it may be appropriate to create a subset of the original data with some types of data, but not others. Thus, we will look at the methods of creating child MVC objects by the class of the parent model data. When the model contains a data frame, the implemented methods for creating a child model are to create a subset model, and to perform multidimensional scaling (MDS). When the model contains an ExpressionSet, the implemented methods for creating a child model are to create a Gene Ontology (GO) graph, to perform gene set enrichment using KEGG pathways as the categories, and to perform gene set enrichment using chromosome location as the categories. Currently, there are no methods to create a child model when the active MVC contains a graph or a gene set enrichment (GSE) model. The hope is that these implemented functions to create a child model (and thus, a child MVC) are only a starting point and that users will extend these options. More information on implementing new methods to create a child MVC is given in Section 14.

For an extended example, please see Sections 5.2.1 and 10.1.3 where the original MVC contains an ExpressionSet model. A GO graph child MVC is created from the original ExpressionSet

model. Then by interacting with a node on a plot of the GO graph in Section 10.1.3, a new `ExpressionSet` model is created that contains only the genes that are annotated at that GO term. Thus, this extended example starts with a `MVC` that contains an `ExpressionSet`, then makes a child `MVC` for the GO graph, and finally makes a grandchild `MVC` that contains an `ExpressionSet` with only the genes annotated at a particular node from the GO graph.

5.1 The Active `MVC` Contains a Data Frame

There are two options for creating a child model from a parent model that contains a data frame and these options are to create a subset model, and to perform MDS. Each of these options for creating a child `MVC` is discussed individually.

These methods (creating a subset or performing MDS) to create child `MVCs` all result in a child model that is also a data frame model and the row names in the child model will be a subset of the row names that are in the parent model. Thus, the child model has a one-to-one relationship with its parent model. The linking functions to relate elements in a child model to elements in a parent model and vice versa, just match row names since the row name in the child model will match the row name in the parent model. Thus, if the row for 'Minnesota' is highlighted in the child model, then the row for 'Minnesota' can also be highlighted in the parent model. This linking of views of different models is performed by the linking functions and is discussed in more detail in Whalen and Gentleman (2006a) and Whalen and Gentleman (2006b). Interacting with views to select data elements is discussed in Section 10.

5.1.1 Create a Subset

Creating a subset from a `MVC` object that contains a data frame is probably the simplest example of creating a child `MVC`. All that is done to create this child `MVC` is determine which rows from the parent `MVC's` model are included in the child model. This subsetting can be performed by any type of boolean statement. For example, the subsetting can be performed by looking for particular values of one of the variables (column names) stored in the data frame. With the USArrests `MVC`, the user may want to create a new `MVC` that contains only rows with values of Assault that are greater than 200.

Using the GUI

First set the active `MVC` to USArrests using the Edit menu (see Section 4). Then go to File menu and choose the Create Child Model menu item or type Ctrl-C. For method to create child model, choose Create Subset Model and for name of the new child model, type USAsub. Finally, click the Create Child Model button. A new window appears asking the user how to subset the data frame model. I will choose Assault as the Variable and greater than as the Relationship and type 200 as the Value. Then click the Done button. Now the USAsub `MVC` has been loaded. To set this new `MVC` as the active `MVC` go to the Edit menu and choose Set Active MVC.

Using the Command Line

To create a subset MVC using the command line, first set USArrests as the active MVC. Then call the function getMethodsToCreateChild to see that creating a subset is an available method to create a child MVC. Finally, call the function createSubset to create the new child MVC. The createSubset function has two parameters: newModelName, which is the name of the new MVC, and rowNames, which are the row names that are included in the new model. For more information on these functions, please see their man pages. The code to create a subset MVC is shown below.

```
> setActiveMVC("USArrests")
> getMethodsToCreateChild()
> createSubset("USAsub", rowNames = rownames(USArrests)[USArrests$Assault >
+     200])
> setActiveMVC("USAsub")
```

5.1.2 Perform Multidimensional Scaling (MDS)

Performing multidimensional scaling (MDS) on a MVC object that contains a data frame model creates a new child MVC object that contains a data frame model. MDS is performed using the cmdscale function on the parent model's data. The user is encouraged to see the man page for cmdscale for more information on MDS. The only information that is needed to perform MDS on the data frame model is the number of dimensions to include in the new child model and the dimensions can be any number between 1 and n (where n is the number of dimensions of the parent model).

Using the GUI

To perform MDS through the GUI, first set the active MVC to USArrests (see Section 4) using the Edit menu. Then go to the File menu and choose Create Child Model or type Ctrl-C. For method to create child model, choose Perform Multidimensional Scaling (MDS) and for name of the new child model, type USAMDS. Then click on the Create Child Model button. A new window appears asking the user how many dimensions to include in the child model. The default value is 2, which is what I will use. Then click the Done button. Now the USAMDS MVC object has been loaded. The user can then set USAMDS as the active MVC to create views of the child model.

Using the Command Line

To perform MDS using the command line, first set USArrests as the active MVC. Then call the function getMethodsToCreateChild to see that performing MDS is an available method to create a child MVC. Finally, call the function performMDS to perform MDS and create a child MVC. The performMDS has three parameters: newModelName, which is the name of the new model, k, which is the number of dimensions, and transpose, which is whether the matrix should be transposed before performing MDS. Currently, the default for transpose is FALSE and this parameter is actually not used in the function definition because it makes linking the child MVC to the parent MVC too difficult if the matrix is transposed (here linking is performed by

matching row names). For more information on these functions, please see their man pages. The code to perform MDS and create a child MVC is shown below.

```
> setActiveMVC("USArrests")
> getMethodsToCreateChild()
> performMDS("USAMDS", k = 2)
> setActiveMVC("USAMDS")
```

5.2 The Active MVC Contains an ExpressionSet

There are currently three options for creating a child model from a parent model that contains an ExpressionSet and these options are to create a GO graph, perform gene set enrichment using chromosome locations as the categories, and to perform gene set enrichment using KEGG pathways as categories. Each of these options for creating a child MVC is discussed individually.

The linking functions that relate elements in the parent model to elements in the child model are more complex now because the child models of an ExpressionSet model have a one-to-many or a many-to-many relationship. For example, the genes in the ExpressionSet model can be annotated at several GO terms and each GO term can have many genes annotated at that term. Thus, when the parent MVC contains an ExpressionSet and the child MVC contains a GO graph, then the elements in two models have a many-to-many relationship and deciding how these models should be properly linked requires some thought. A full discussion of these ideas can be found in Whalen and Gentleman (2006a) and Whalen and Gentleman (2006b). As each method for creating a child model is discussed in the following paragraphs, we will also describe how the linking functions work for this particular child model to the ExpressionSet parent model.

5.2.1 Create a GO Graph

Creating a GO graph from a parent model requires that the parent model contain genetic data, particularly Affymetrix microarray data. This requirement is filled by a parent model that contains an ExpressionSet. With an ExpressionSet model the Affymetrix identifiers used to create the GO graph are taken from the row names of the exprs slot of the ExpressionSet (this information can also be obtained using the geneNames function).

When a GO graph is created from an ExpressionSet model, the two linking functions are defined: one that converts information from the parent to the child and one that converts information from the child to the parent. The toParent function takes a GO term that was just updated from the child model and converts it into the genes that are annotated at that term. The update information is the sent to the parent model with the genes that could be updated. Depending on what the update method is, the genes in the parent may also be updated. For instance, if the update in the child model colors or highlights a GO term, then the linked genes in the parent model are also colored or highlighted. However, if the update in the child model

was to hide a GO term, then the linked genes will only be hidden if all the GO terms these genes are annotated at are also hidden.

The fromParent function takes a gene that was just updated from the parent model and converts it into the GO terms that this gene is annotated at. The update information is sent to the child model with the GO terms that could be updated. Again, the message sent to the child depends on the update method that just occurred. If the update in the parent model is to color or highlight a gene, then the linked GO terms in the child model are also colored or highlighted. However, if the update in the parent model was to hide a gene, then the linked GO terms are only hidden if all genes that are annotated at them are also hidden.

More information on how to update the data through views is given in Section 10.

Using the GUI

First set the CLL `MVC` to be the active `MVC` using the Edit menu (see Section 4). Now go to the File menu and choose the Create Child Model menu item or type Ctrl-C. Choose Create GO Graph as the method to create child model and type CLLgo as the name of the new child model. If the annotation slot is an empty string in the `ExpressionSet`, then a new window appears asking what the Affymetrix Chip Type is. Now several libraries are loaded that are necessary to convert Affymetrix identifiers into GO terms. Finally a window opens that asks the user to choose the Affymetrix identifiers that will be used to create the GO graph. I will use all the Affymetrix identifiers that passed the specific filter so in the Subset by Model Variable drop down box, I will choose sFiltertBH. Then I will use the Select All button to select all the Affymetrix identifiers (these are the Affymetrix identifiers that passed the specific filter - there are 78 of them). I will leave the Ontology Type drop down box as MF, indicating that I want the molecular function GO graph. Finally I'll click the Done button. Creating the GO graph may take a while, but the window with the selected Affy Ids disappears when the new GO graph model has been created.

Now the user can set the CLLgo `MVC` as the active `MVC` to create views of the GO graph. The user may also want to create the model graph (described in Section 11.2) to see how the new `MVC` object is related to other loaded `MVCs`.

Using the Command Line

The code to create a GO graph `MVC` from a parent `MVC` that contains an `ExpressionSet` model is shown below. The function getMethodsToCreateChild lets the user see which methods exist to create a child `MVC` from the active `MVC`. Next to create the GO graph from the CLL `MVC`, the function createGoGraph is called. The createGoGraph function has five parameters: newModelName, annotationSlot, ontology, affyids, and mVar. Please see the man pages for these functions for more information.

```
> setActiveMVC("CLL")
> getMethodsToCreateChild()
> createGoGraph("CLLgo", annotationSlot = "hgu95av2", ontology = "MF",
+     mVar = "sFiltertBH")
```

```
> setActiveMVC("CLLgo")
```

5.2.2 Perform Gene Set Enrichment Using Chromosome Location as Categories

Creating a gene set enrichment (GSE) model involves using gene test statistics and genes in categories information (what we call the incidence matrix) to calculate gene set statistics. Here we use chromosome locations to create the gene sets. For a more detailed explanation of gene set enrichment and how it is used in the *iSNetwork* package, please see Whalen and Gentleman (2006b). The chromosome locations are determined using the chrCats and users are referred to the man page for that function for full information on how the chromosome locations are determined.

Here, we have a many-to-many relationship between the elements in the parent `MVC` (the genes in the `ExpressionSet` model) and the elements in the child `MVC` (the gene sets in the `GSE` model). A gene can be on many locations (for example, the gene can be at the following locations: 17, 17p, 17p1, and 17p13) and each location can have more than one gene. Thus, the linking functions, toParent and fromParent, must determine how to relate these models when an update occurs.

The toParent function takes a gene set that was just updated from the child model and converts it into the genes that belong to that gene set. The update information is the sent to the parent model with the genes that could be updated. Depending on what the update method is, the genes in the parent may also be updated. For instance, if the update in the child model colors or highlights a gene set, then the linked genes in the parent model are also colored or highlighted. However, if the update in the child model was to hide a gene set, then the linked genes will only be hidden if all the gene sets these genes are annotated at are also hidden.

The fromParent function takes a gene that was just updated from the parent model and converts it into the gene sets that this gene belongs to. The update information is sent to the child model with the gene sets that could be updated. Again, the message sent to the child depends on the update method that just occurred. If the update in the parent model is to color or highlight a gene, then the linked gene sets in the child model are also colored or highlighted. However, if the update in the parent model was to hide a gene, then the linked gene sets are only hidden if all genes that belong to them are also hidden.

More information on how to update the data through views is given in Section 10.

Using the GUI

First set the CLL `MVC` to be the active `MVC` using the Edit menu (see Section 4). Now go to the File menu and choose the Create Child Model menu item or type Ctrl-C. Choose Perform Gene Set Enrichment using Chromosome as the method to create child model and type CLLgseChr as the name of the new child model. If the annotation slot is an empty string in the `ExpressionSet`, then a new window appears asking what the Affymetrix Chip Type is. Now several libraries are loaded that are necessary to calculate which genes belong to which gene sets. Finally a window opens that asks the user to choose the Affymetrix identifiers that

will be used to create the gene set enrichment model. I will use all the Affymetrix identifiers that passed the nonspecific filter so in the Subset by Model Variable drop down box, I will choose nsFilter. Then I will use the Select All button to select all the Affymetrix identifiers (these are the Affymetrix identifiers that passed the nonspecific filter - there are 6313 of them). I will use Disease as the Factor for t-test (the t-test statistic is used as the gene test statistic; please see Whalen and Gentleman (2006b) for more information), and I will require that a gene set have at least 5 genes (set to 5 in the Number of Genes in Gene Set text box). Finally I'll click the Done button. Creating the gene set enrichment model may take a while, but the window with the selected Affy Ids disappears when the new gene set enrichment model has been created.

Now the user can set the CLLgseChr `MVC` as the active `MVC` to create views of the gene set enrichment model. The user may also want to create the model graph (described in Section 11.2) to see how the new `MVC` object is related to other loaded `MVCs`.

Using the Command Line

The code to create a gene set enrichment model (using chromosome location as the categories) `MVC` from a parent `MVC` that contains an `ExpressionSet` model is shown below. The function getMethodsToCreateChild lets the user see which methods exist to create a child `MVC` from the active `MVC`. Next to create the gene set enrichment model from the CLL `MVC`, the function createChrLocGSE is called. The createChrLocGSE function has six parameters: newModelName, fac, annot, affyids, mVar, and noGenes. Please see the man pages for these functions for more information.

```
> setActiveMVC("CLL")
> getMethodsToCreateChild()
> createChrLocGSE(newModelName = "CLLgseChr", fac = "Disease",
+     mVar = "nsFilter")
> setActiveMVC("CLLgseChr")
```

5.2.3 Perform Gene Set Enrichment Using KEGG Pathways as Categories

Now we use KEGG Pathways as categories to create the gene set test statistics. Please see Whalen and Gentleman (2006b) for more information on gene set enrichment analysis and how it is implemented in this package. Again, we have a many-to-many relationship between the elements in the parent `MVC` (the genes in the `ExpressionSet` model) and the elements in the child `MVC` (the gene sets in the `GSE` model). A gene can be in many pathways and a pathway can contain many genes. Please see Section 5.2.2 for more information on the linking functions that are used to connect genes to gene sets.

Using the GUI

First set the CLL `MVC` to be the active `MVC` using the Edit menu (see Section 4). Now go to the File menu and choose the Create Child Model menu item or type Ctrl-C. Choose Perform Gene Set Enrichment using KEGG as the method to create child model and type

CLLgseKEGG as the name of the new child model. If the annotation slot is an empty string in the ExpressionSet, then a new window appears asking what the Affymetrix Chip Type is. Now several libraries are loaded that are necessary to calculate which genes belong to which gene sets. Finally a window opens that asks the user to choose the Affymetrix identifiers that will be used to create the gene set enrichment model. I will use all the Affymetrix identifiers that passed the nonspecific filter so in the Subset by Model Variable drop down box, I will choose nsFilter. Then I will use the Select All button to select all the Affymetrix identifiers (these are the Affymetrix identifiers that passed the nonspecific filter - there are 6313 of them). I will use Disease as the Factor for t-test (the t-test statistic is used as the gene test statistic; please see Whalen and Gentleman (2006b) for more information), and I will require that a gene set have at least 5 genes (set to 5 in the Number of Genes in Gene Set text box). Finally I'll click the Done button. Creating the gene set enrichment model may take a while, but the window with the selected Affy Ids disappears when the new gene set enrichment model has been created.

Now the user can set the CLLgseKEGG MVC as the active MVC to create views of the gene set enrichment model. The user may also want to create the model graph (described in Section 11.2) to see how the new MVC object is related to other loaded MVCs.

Using the Command Line

The code to create a gene set enrichment model (using KEGG pathways as the categories) MVC from a parent MVC that contains an ExpressionSet model is shown below. The function getMethodsToCreateChild lets the user see which methods exist to create a child MVC from the active MVC. Next to create the gene set enrichment model from the CLL MVC, the function createKEGGGSE is called. The createKEGGGSE function has six parameters: newModelName, fac, annot, affyids, mVar, and noGenes. Please see the man pages for these functions for more information.

```
> setActiveMVC("CLL")
> getMethodsToCreateChild()
> createKEGGGSE(newModelName = "CLLgseKEGG", fac = "Disease", mVar = "nsFilter")
> setActiveMVC("CLLgseKEGG")
```

5.3 The Active MVC Contains a Graph

When the active MVC contains a graph model, then there are currently no methods implemented to create a child MVC. This can be seen by calling the function, getMethodsToCreatChild, when the active MVC contains a graph. This can also be seen by clicking on the Create Child Model menu item under the File menu on the control window and the control window is a blank screen, indicating that there are no currently implemented methods to create a child MVC.

One obvious possibility to create a child MVC from a graph MVC is to create a subgraph child model. To implement new methods to create child MVCs, please see Section 14.

5.4 The Active MVC Contains a GSE

When the active MVC contains a gene set enrichment (GSE) model, then there are currently no methods implemented to create a child MVC. This can be seen by calling the function, getMethodsToCreatChild, when the active MVC contains a GSE. This can also be seen by clicking on the Create Child Model menu item under the File menu on the control window and the control window is a blank screen, indicating that there are no currently implemented methods to create a child MVC.

One possibility to create a child MVC from a GSE MVC is to create an ExpressionSet model that contains only the genes that are located in one gene set (for example all genes in the apoptosis biological process). To implement new methods to create child MVCs, please see Section 14.

6 Creating Views

The views that have been currently implemented include scatter plots, spreadsheets, graph plots, heatmaps, and qq-plots. If the active MVC has a model of type data.frame, then the available views are scatter plots and spreadsheets. If the active MVC has a model of type graph, then the available views are graph plots and spreadsheets (the spreadsheet displays node information). If the active MVC has a model of type ExpressionSet, then the only available view is a heatmap. Finally, if the active MVC has a model of type GSE for a gene set enrichment model, then the available views are qq-plots and spreadsheets. More details of these views for each type of model is given in the following sections. All of these views are interactive, which is discussed in Section 10.

Currently, the spreadsheet view is available for several types of models. The spreadsheet view presents an Excel-type format to display two-dimensional data. This view can be resorted by clicking on the column header buttons. If a column header is clicked on by the user, the spreadsheet is rearranged so that the column is sorted alphabetically. Note that sorting alphabetically will give unexpected results if the values stored in that column are numeric. A future goal is to change the sorting function based on the values stored in the column.

7 Views of Data Frames

If the active MVC object has a model of class data.frame, then the user can currently create two types of views of this data and these views are spreadsheets and scatter plots. The Display menu on the control window has two menu items, View Data and Plot Data, when the active MVC has a model of class data.frame.

7.1 Creating Spreadsheets

After loading data using the methods described in Section 3, the data can be viewed in an Excel-type format if the active `MVC` contains a `data.frame`, a `graph`, or a `GSE` (which represents gene set enrichment) model. Currently, a `data.frame` model can be viewed in only one window. For example, if the USArrests data frame is being shown in a window, it can not be shown in a different window.

Even though each model can be shown in only one window, there is no limit on how many data sets can be shown in separate windows. That means you can have one window showing USArrests and another window showing a different `data.frame` data set, and so on. Recall that views can only be created of the active `MVC` so to create a view of a different data set, the active `MVC` must be reset. This operation was discussed in Section 4.

A few things to note about the spreadsheet view. The data can not be changed through the view data windows. However, the spreadsheet is sortable. If the user wants to sort the spreadsheet based on one of the columns, the user can click on the column header and the spreadsheet will be alphabetically sorted based on that column. Be aware that sorting alphabetically will not always be what users want, particularly when the column contains numbers, but currently the sorting function cannot be changed.

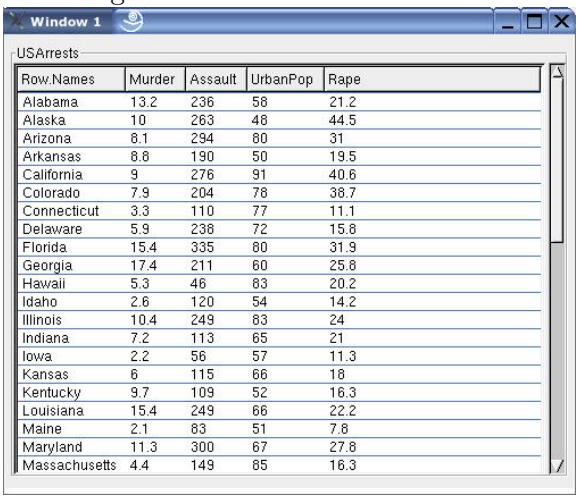
7.1.1 Creating a Spreadsheet Through the GUI

To view the data, highlight the Display menu and select the View Data menu item. Alternatively, typing Ctrl-D activates the View Data menu item. The control window now appears as shown in Figure 5 after selecting the View Data menu item.

Figure 5: View the Data

Now click the Show USArrests button. Another window appears that shows the data frame data. The new window showing the USArrests data frame is shown in Figure 6.

Figure 6: View USArrests Data



Row.Names	Murder	Assault	UrbanPop	Rape
Alabama	13.2	236	58	21.2
Alaska	10	263	48	44.5
Arizona	8.1	294	80	31
Arkansas	8.8	190	50	19.5
California	9	276	91	40.6
Colorado	7.9	204	78	38.7
Connecticut	3.3	110	77	11.1
Delaware	5.9	238	72	15.8
Florida	15.4	335	80	31.9
Georgia	17.4	211	60	25.8
Hawaii	5.3	46	83	20.2
Idaho	2.6	120	54	14.2
Illinois	10.4	249	83	24
Indiana	7.2	113	65	21
Iowa	2.2	56	57	11.3
Kansas	6	115	66	18
Kentucky	9.7	109	52	16.3
Louisiana	15.4	249	66	22.2
Maine	2.1	83	51	7.8
Maryland	11.3	300	67	27.8
Massachusetts	4.4	149	85	16.3

7.1.2 Creating a Spreadsheet Through the Command Line

To create a spreadsheet of the active `MVC's` data set through the command line, call the function, createSpreadsheet. Currently, spreadsheet views can be created if the active `MVC` has a model of class `data.frame`, `graph`, and `GSE`, where `GSE` represents a gene set enrichment data set. Views of `graphs` and `GSEs` are discussed in Section 8 and 9.2, respectively.

Example code using this function is shown below. It takes no parameters since it only creates a spreadsheet view of the active `MVC's` data. Please see the man page for this function for more information.

```
> setActiveMVC("USArrests")
> createSpreadsheet()
```

7.2 Creating Scatter Plots

Once a model is loaded, the user can create scatter plots of `data.frame` data through the GUI or through the command line function, createSPlot. Remember that only the active `MVC` object's data can be plotted.

Unlike viewing the data where each data set can only appear in one spreadsheet window, users can create as many scatter plots of each `data.frame` data set as they like. Each new scatter plot is presented in a new window.

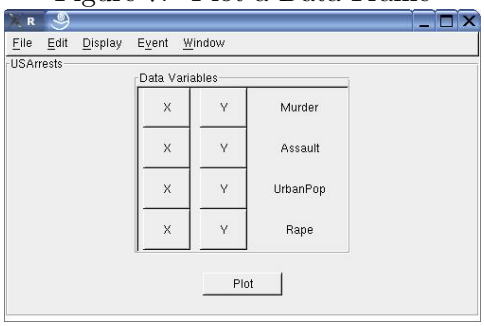
Recall from Section 3 that each `MVC` object contains one model and all views of this model. Thus, plots that are based on the same data are stored in the same `MVC` object and they are linked. This idea is discussed further in Section 10.

7.2.1 Creating a Scatter Plot Through the GUI

To plot a data frame, highlight the Display menu and select the Plot Data menu item. Alternatively, the user can press Ctrl-P to activate the Plot Data menu item.

After activating the Plot Data menu item, the control window shows a frame called Data Variables, which contains toggle buttons to set the X and Y variables for the scatter plot using the column names from the active `MVC` object's data set. The control window now appears as shown in Figure 7.

Figure 7: Plot a Data Frame



The variables available for plotting USArrests data are Murder, Assault, UrbanPop, and Rape. To learn more about these variables , the user can type help ("USArrests") at the R prompt. As an example, click the X button for Murder and click the Y button for Assault, and then click the Plot button. A scatter plot of Assault vs. Murder appears in a new window, as shown in Figure 8.

Only one X button and Y button can be highlighted at a time because only two-way scatter plots can be made at this time.

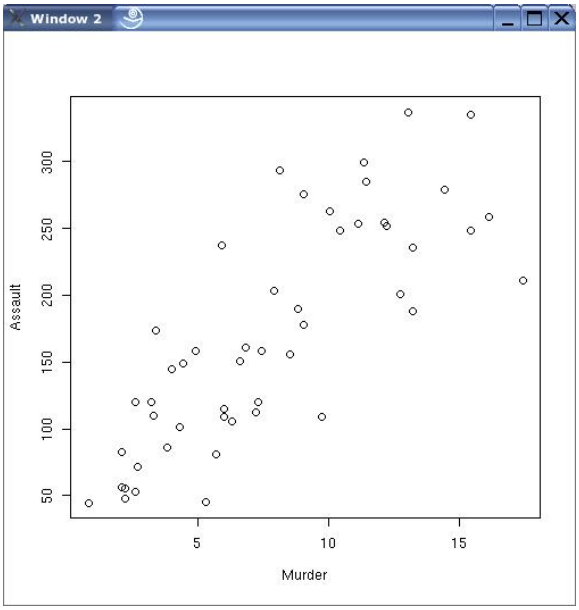
To create another plot of USArrests data, click the X button for UrbanPop and the Y button for Rape, and then click the Plot button. Now there are two plots of USArrests data. One of Assault vs. Murder and one of Rape vs. UrbanPop.

7.2.2 Creating a Scatter Plot Through the Command Line

To create a scatter plot of the active `MVC` object's data set through the command line, call the function, createSPlot. The example code below creates a scatter plot with Murder as the X variable and Rape as the Y variable. Murder and Rape are the names of columns from the USArrests data. Please see the man page for this function for more information.

```
> setActiveMVC("USArrests")
> createSPlot(varx = "Murder", vary = "Rape")
```

Figure 8: Scatter Plot of Assault vs. Murder for USArrests



8 Views of Graphs

Currently, two views of graphs can be created: a graph plot and a spreadsheet view of node data. When the active `MVC` object has a model of class `graph`, then the Display menu on the control window has two menu items named Plot Graph and View Node Data.

8.1 Plotting a Graph

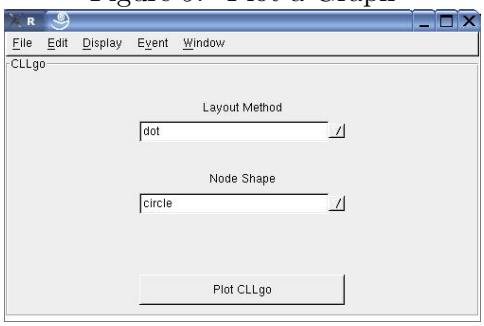
Once a graph data set is loaded, the user can create plots of `graph` data through the GUI or through the command line function, createGraphPlot. Multiple plots of `graph` objects can be created if the user wants to view the `graph` with different layout types or node shapes. Remember that only the active `MVC` object's data can be plotted.

8.1.1 Plotting a Graph Through the GUI

To plot a `graph`, first make sure that the active `MVC` contains a graph data set. Set the active `MVC` to be CLLgo, which was created in Section 5.2.1. Then highlight the Display menu and select the Plot Graph menu item. Alternatively, the user can press Ctrl-P to activate the Plot Graph menu item. The control window now appears as shown in Figure 9 after selecting the Plot Graph menu item.

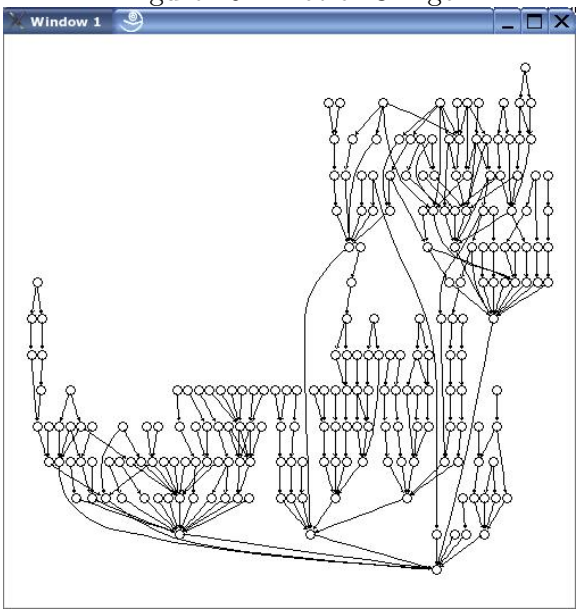
For plotting the graph the user can choose the Layout Method and the Node Shape, as shown in Figure 9. The options for the Layout Method are 'dot', 'neato', and 'twopi'. The different

Figure 9: Plot a Graph



layout methods are algorithms based on optimizing a particular aspect of the layout Gansner and North (2000), such as minimizing edge length or minimizing edge crossings. Please see the help for layoutType for more information. The options for the Node Shape are 'circle', 'ellipse', and 'rectangle'. As an example, choose a Layout Method of 'dot' and a Node Shape of 'circle' and click the Plot CLLgo button. A plot of the graph appears in a new window, as shown in Figure 10.

Figure 10: Plot of CLLgo



Other plots of this graph can be created if the user wants to look at the graph with different layouts or node shapes.

8.1.2 Plotting a Graph Through the Command Line

To create a graph plot of the active MVC object's data set through the command line, call the function, createGraphPlot. The example code below creates a plot of the CLLgo model with a layout of "twopi" and a node shape of "ellipse". Please see the man page for createGraphPlot for more information on this function.

```
> setActiveMVC("CLLgo")
> createGraphPlot(layout = "dot", nodeShape = "circle")
```

8.2 Viewing Node Data

The other view for a graph model is to show the node attributes in a spreadsheet. On the control window, this option is available under the Display menu as the View Node Data menu item. To create the node attribute spreadsheet through the GUI, the user can call the createSpreadsheet function. Note that if the graph does not have node attributes, then the spreadsheet view cannot be created. For more information on the spreadsheet view, please see Section 7.1.

9 Views of ExprSets

Currently, the only view that can be created of a ExpressionSet data set is a heatmap. When the active MVC object has a model of class ExpressionSet, then the Display menu on the control window has one menu item named Plot Heatmap.

9.1 Creating a Heatmap

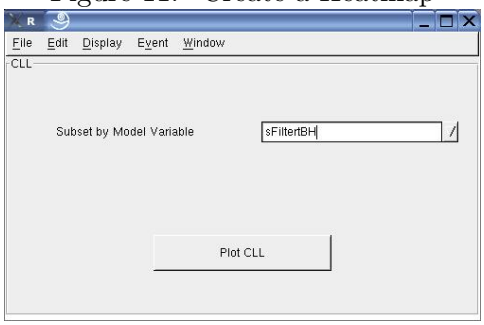
Once a data set is loaded, the user can create plots of ExpressionSet data through the GUI or through the command line function, createHeatmap.

9.1.1 Creating a Heatmap Through the GUI

Set the active MVC to be CLL, which was loaded in Section 3.2. Then highlight the Display menu and select the Plot Heatmap menu item. Alternatively, the user can press Ctrl-P to activate the Plot Heatmap menu item. The control window now appears as shown in Figure 11 after selecting the Plot Heatmap menu item.

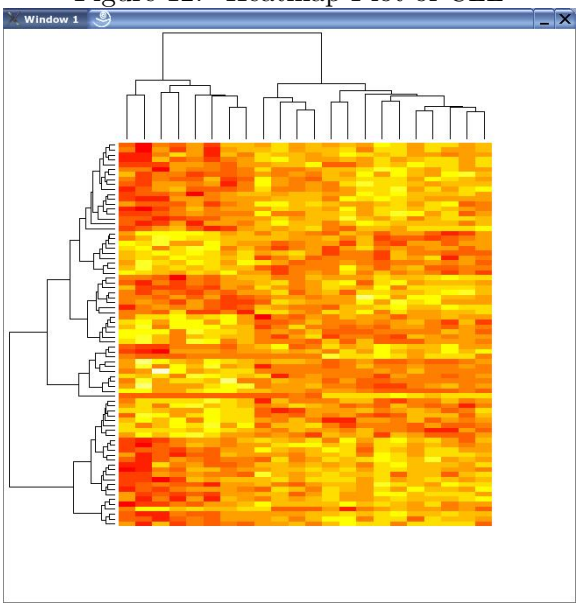
The user can decide to plot the whole model or can subset the model by using one of the model variables. This requires that the model variable be a boolean variable that indicates how the model should be subset for the heatmap view. In our example, we will subset the model using the sFiltertBH model variable, which was loaded in Section 3.3. The full CLL model has 12,625 genes, which would make a very large heatmap. By using the sFiltertBH variable to subset the

Figure 11: Create a Heatmap



CLL model, only 78 genes will be plotted on the heatmap. Now click the Plot CLL button. A new window appears that shows the heatmap plot, which is shown in Figure 12.

Figure 12: Heatmap Plot of CLL



9.1.2 Creating a Heatmap Through the Command Line

To create a heatmap plot of the active `MVC` object's data set through the command line, call the function, createHeatmap. The only parameter for this function is mVar, which lets the user subset the model using a model variable. This model variable must be a boolean to indicate how the genes in the model should be subset. The example code below creates a heatmap of a subset of the CLL model. Please see the man page for createHeatmap for more information on this function.

```
> setActiveMVC("CLL")
> createHeatmap(mVar = "sFiltertBH")
```

9.2 Views of Gene Set Enrichment (GSE) Models

Currently, there are two possible views for a gene set enrichment model: a qq-plot and a spreadsheet view. Gene set enrichment models can only be created by creating a child model from an `ExpressionSet` model, which was discussed in Section 5. We will show the views for a gene set enrichment model that was created from the CLL model here. This gene set enrichment model used the genes from the CLL model that passed the nonspecific filter (nsFilter model variable), used KEGG pathways as the categories, only included pathways that had at least 5 genes, and used Disease as the phenotype factor for the t-test. Please see Section 5 for more information on creating this model, called CLLgseKEGG.

When the active `MVC` object has a model of class `GSE`, then the Display menu on the control window has two menu items, Create QQ Plot and View Data.

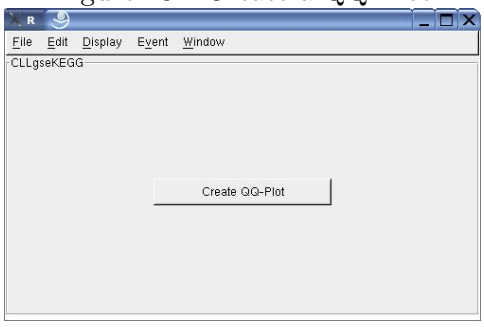
9.3 Creating a QQ Plot

Once the `GSE` model has been created from an existing `ExpressionSet` model and set as the active `MVC`, then the user can create qq-plots of the gene set test statistics through the GUI or through the command line function, createQQplot.

9.3.1 Creating a QQ Plot Through the GUI

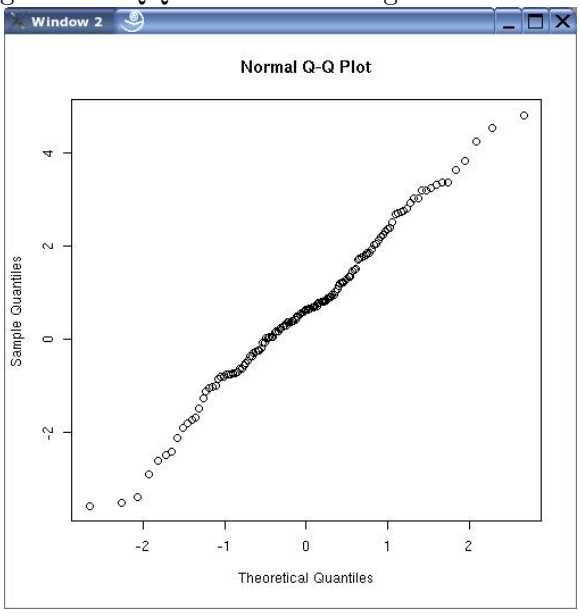
Set the active `MVC` to be CLLgseKEGG, which is created in Section 5. Then highlight the Display menu and select the Create QQ Plot menu item. Alternatively, the user can press Ctrl-P to activate the Create QQ Plot menu item. The control window now appears as shown in Figure 13 after selecting the Create QQ Plot menu item.

Figure 13: Create a QQ-Plot



Now click the Create QQ-Plot button. A new window appears that shows the qq-plot, which is shown in Figure 14.

Figure 14: QQ-Plot of the CLLgseKEGG model.



9.3.2 Creating a QQ Plot Through the Command Line

To create a qq-plot of the active `MVC` object's data set through the command line, call the function, createQQplot. This function has no parameters. Please see the man page for createQQplot for more information on this function.

```
> setActiveMVC("CLLgseKEGG")
> createQQplot()
```

9.4 Viewing the `GSE` Data

The other view for a `GSE` model is to show the gene set attributes in a spreadsheet. On the control window, this option is available under the Display menu as the View Data menu item. To create the gene set attribute spreadsheet through the GUI, the user can call the createSpreadsheet function. For more information on the spreadsheet view, please see Section 7.1.

10 Setting Callback Functions to Events

Having interactive views was one of the goals discussed in Section 1.1. Creating interactive applications requires three things: some user action or input, which is referred to as an event; a response to that action, which is executed by a callback function; and a method that connects the event to the callback function, which is referred to as the signal handler. These three steps are ordered as follows: an event occurs, the event causes a signal to be emitted that is caught by the signal handler, and then the signal handler calls the callback function, which results in the response to the user action. A flowchart of these steps is shown below in Figure 15.

Figure 15: Flow Chart of the Response to an Event

In an example given in Section 1.1, a point was colored red when the user clicked on that point in the view. In this example of interactivity, the event is the mouse button click and the response, which is caused by a callback function, is to color a point red. The signal handler, which is the method that connects the event to the callback function, is not noticed by the user.

In the *iNetwork* package, another goal is to have a flexible design. Thus, we did not want the response to an event to be fixed. We wanted to allow the user to decide what the response to an event should be. For example, the user may initially want a point to be colored when a mouse click occurs over that point and later the user may decide that a mouse click event over a point should cause the point to be hidden. Recall that the response to an event is determined by the callback function. Thus, for flexibility in the *iSNetwork* package, the user can set the callback function for certain events. This lets the user decide what the response to an event is.

Currently, the four events that the user can set callback functions to are mouse over, left button click, middle button click, and right button click. This means that the user can decide what happens in response to any of these four events. Each event is discussed in more detail in the following sections.

One important thing to notice is that these callback functions can only be set for views of the active MVC object. Thus, when setting the callback function to an event through either the GUI or through command line functions, these responses will only be set for views of the active MVC. However, if views of different MVC's have been created and callback functions were previously set for these views, then when the user interacts with these views of the non-active MVC, the previously set callback function will be called.

An example will help explain these concepts. Suppose a user had two loaded MVC objects, one was called "USArrests" and one was called "USJudgeRatings". Next suppose that the active MVC object is "USArrests", two scatter plots of this model are created, and the callback function for the left button click event is to color a point blue. Then the user sets the active

MVC to be "USJudgeRatings", creates two new scatter plots of this model (so there are now four scatter plots: two of "USArrests" and two of "USJudgeRatings"), and sets the callback function for the left button click event to highlight a point. If the user now clicks the left button over a point in one of the views of "USArrests", then this point will be colored blue (even though this view is not depicting the active MVC). If the user clicks the left button over a point in one of the views of "USJudgeRatings", then this point will be highlighted (recall this is a view of the active MVC).

Also, the potential responses to an event depend on the class of the active `MVC's` model. Recall from Sections 7, 8, 9, and 9.2, that for different types of models, only certain views make sense to display the data. For a data frame model, the user can create a spreadsheet or a scatter plot view, for a graph model, the user can create a plot of the graph or a spreadsheet, for an `ExpressionSet` model, the user can create a heatmap, and for a `GSE` model, the user can create a qq-plot or a spreadsheet. In this section, we are talking about setting the response to user interaction with a view. Thus, it makes sense to have different potential responses to an event depending on what type of view is shown. For example, if the active `MVC` object contains a data frame, then one potential response to a left button click on a scatter plot is to color a point whereas if the active `MVC` object contains a graph, then a potential response to a left button click on a plot is to color a node.

10.1 Setting the Left Button Click

The potential responses to a left button click depend on the class of the active `MVC's` model. Thus, if the active `MVC's` model is of class `dfModel` (which means the model contains a data frame), then the potential responses to a left button click are color a point, hide a point, highlight a point, or no action. If the active `MVC's` model is of class `graphModel` (which means the model contains a graph), then the potential responses to a left button click are color a node, hide a node, highlight a node, create a heatmap, or no action. If the active `MVC's` model is of class `exprModel` (which means the model contains an `ExpressionSet`), then the potential responses to a left button click are highlight a gene, color a gene, hide a gene, or no action. If the active `MVC's` model is of class `GSE` (which means the model contains gene set enrichment data), then the potential responses to a left button click are color a point, hide a point, highlight a point, or no action.

10.1.1 Left Button Click Event for a Data Frame

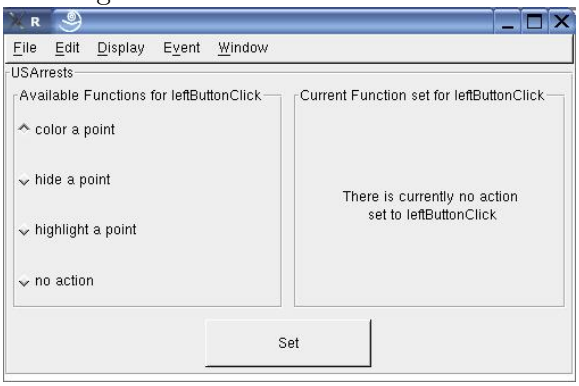
As mentioned previously, the currently available responses for a left button click event when the active `MVC` object's model is a data frame are color a point, hide a point, highlight a point, or no action.

Using the GUI

To see the callback functions that can be set to the left button click event when the active `MVC` contains a data frame, first set the active `MVC` to USArrests and then highlight the Event

menu and select the Set Left Button Click menu item. Alternatively, the user can press Ctrl-B to activate the Set Left Button Click menu item. The control window now looks as shown in Figure 16.

Figure 16: Setting the Left Button Click Event for a Data Frame



Using the Command Line

To see the callback functions that can be set to the left button click event, call the function, getDescForEvent, with a parameter of "leftButtonClick". This function returns the descriptions of all possible callback functions for a certain event. It is this character string (the description) that is used as the shortName parameter in the setCallFunc function.

```
> setActiveMVC("USArrests")
> getDescForEvent("leftButtonClick")
```

Coloring a Point

We will look at each callback function one at a time. First, let's set the left button click event to color a point.

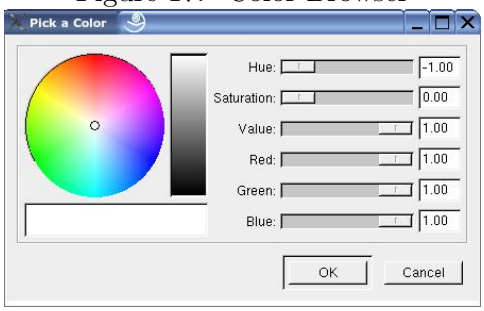
Using the GUI

To do this using the GUI, choose the "color a point" radio button and click the Set button (after choosing the Set Left Button Click menu item under the Event menu). A new window appears that has a color browser to allow the user to choose a color that is used to color the points. Pick a color on the color wheel and click the Ok button. The window with the color wheel is shown in Figure 17.

Using the Command Line

To set the left button click event to color a point using the command line, call the function, setCallFunc, with "leftButtonClick" as the first parameter and "color a point" as the second parameter. The default value for the color is black. If the user wants a different color, then call the function, setColor. To see the current color, call the function, getColor.

Figure 17: Color Browser

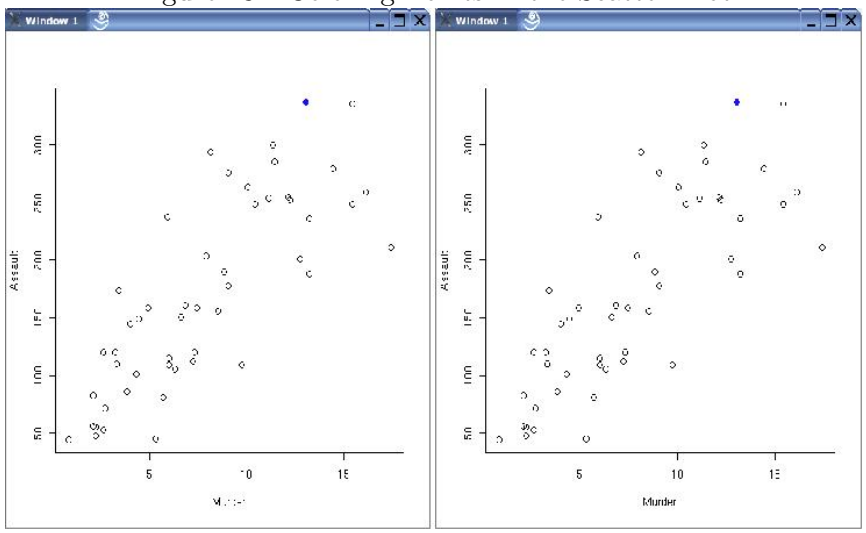



```
> setActiveMVC("USArrests")
> setCallFunc("leftButtonClick", "color a point")
> setColor("blue")
```

Coloring a Point

Now click with the left button on a point in the scatter plot of Assault vs. Murder. Note that this plot was created in Section 7.2. The point that was just clicked is filled in with the new color (I have used blue as shown in Figure 18). Also, a point on the scatter plot of Rape vs. UrbanPop is also filled in with the new color because these two points are from the same row of the data frame. This can be seen by viewing the "USArrests" data in a spreadsheet as was shown in Section 7.1. If the spreadsheet window for "USArrests" is open, one row is selected that corresponds with the point that was just clicked with the left button. Selecting a row in a spreadsheet view is considered the same as clicking a point with the left button over a scatter plot. Please see Section 10.1.2 for more information about selecting a row in a spreadsheet.

Figure 18: Coloring Points in the Scatter Plot



This behavior shows how the scatter plots are linked. Even though a point on the Assault vs. Murder plot was clicked, all plots that are based on the "USArrests" data frame are updated. All points that correspond to that row in the data frame now have the new color. To keep the plots synchronized, some plotting information is stored with the data.

Now, if the user clicks on a point in the scatter plot of Rape vs. UrbanPop, both plots show two blue points each. To choose a different color using the GUI, the user needs to again choose the "color a point" radio button and click the Set button, which causes the window with the color wheel to reappear. To choose a different color using the command line, call the function, setColor.

Hiding a Point

Next, let's set the left button click event to hide a point.

Using the GUI

To do this choose the "hide a point" radio button and click the Set button on the control window (after choosing the Set Left Button Click menu item under the Event menu).

Using the Command Line

To set the left button click event to hide a point using the command line, call the function, setCallFunc, with "leftButtonClick" as the first parameter and "hide a point" as the second parameter.

Hiding a Point

Now click with the left button on a point in the scatter plot of Assault vs. Murder. The point that was just clicked will disappear. A point on the scatter plot of Rape vs. UrbanPop also disappears because these two points are from the same row of the data frame. If a spreadsheet of the "USArrests" data is open, then the row corresponding to the point that was just hidden is selected. To make the point reappear, unselect that row in the spreadsheet. The hidden point now reappears on the scatter plots. Thus, hiding a point is a property that can be toggled on and off. The user can also click with the left button over the spot where the point used to be and the point will reappear.

Another way to show hidden values is to use the Show Hidden Values menu item under the Edit menu (or type Ctrl-H). A new window appears that allows the user to select which hidden values should be shown. With one point hidden, the new window looks as shown in Figure 19. This menu item is also discussed in Section 11.3.

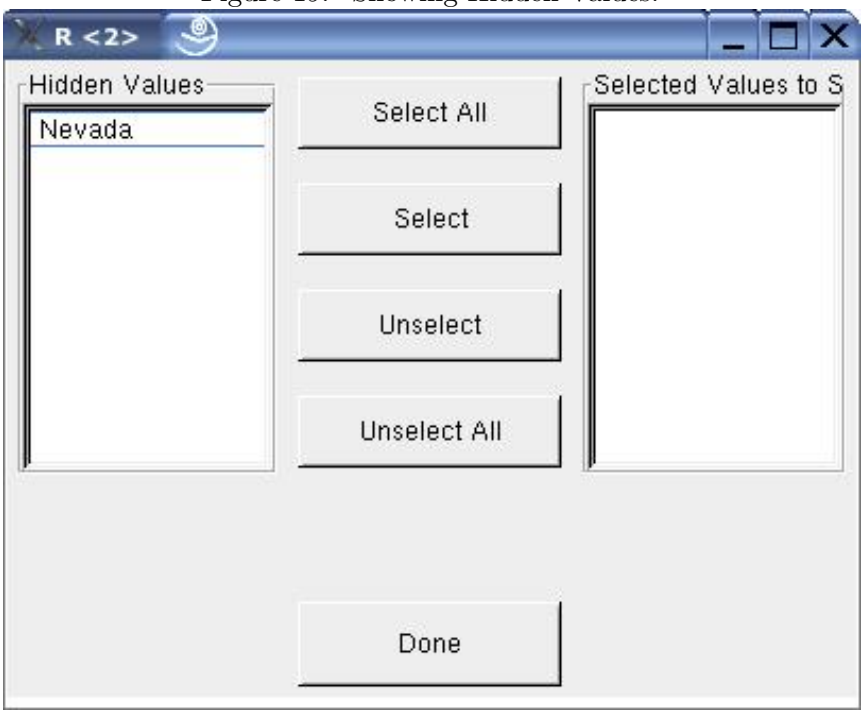
Now select Nevada and click the Done button. Now this point will reappear on the scatter plots.

Highlighting a Point

Next, let's set the left button click event to highlight a point.

Using the GUI

Figure 19: Showing Hidden Values.



To do this choose the "highlight a point" radio button and click the Set button on the control window (after choosing the Set Left Button Click menu item under the Event menu).

Using the Command Line

To set the left button click event to highlight a point using the command line, call the function, setCallFunc, with "leftButtonClick" as the first parameter and "highlight a point" as the second parameter.

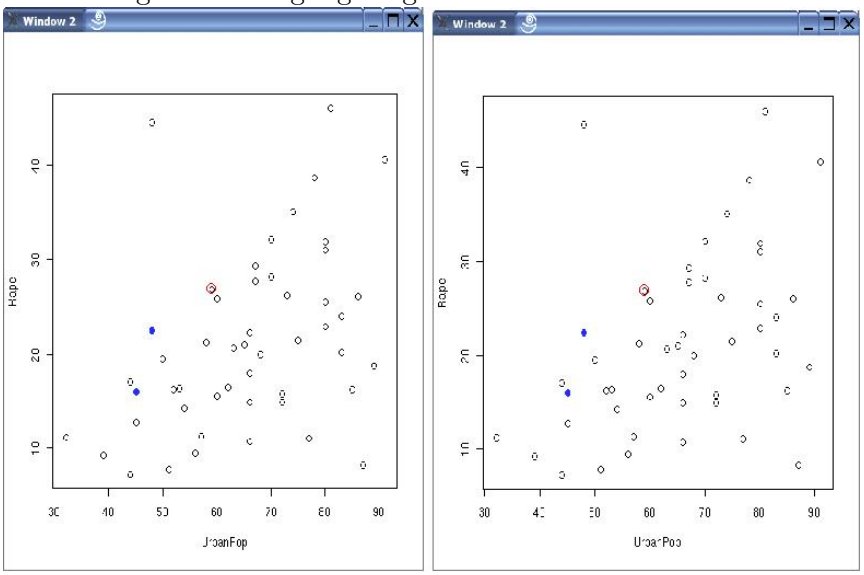
Highlighting a Point

Now click with the left button on a point in the scatter plot of Assault vs. Murder. The point that was just clicked is highlighted (i.e. a red circle appears around that point on the scatter plot). Also, a point on the scatter plot of Rape vs. UrbanPop is highlighted because these two points are from the same row of the data frame. If a spreadsheet of the "USArrests" data is open, then the row corresponding to the point that was just highlighted is selected. To turn off the highlighting, unselect that row in the spreadsheet or click with the left button again over that point. Now the highlighting disappears. Thus, highlighting a point is also a property that can be toggled on and off.

Figure 20 shows a point that has been highlighted on the scatter plots.

No Action

Figure 20: Highlighting Points in the Scatter Plot



The last option and the default is to have no response to a left button click event. When the first view of the active data set is created, there is no response to a left button click event. If the user decides to color a point, hide a point, or highlight a point in response to a left button click, the user can always reset the response to be nothing when a left button click happens.

Notice also that when there is no response to a left button click this means that there is also no response to selecting or unselecting a row in the spreadsheet view.

Using the GUI

To do this choose the "no action" radio button and click the Set button on the control window (after choosing the Set Left Button Click menu item under the Event menu).

Using the Command Line

To set the left button click event to have no response using the command line, call the function, setCallFunc, with "leftButtonClick" as the first parameter and "" as the second parameter.

10.1.2 Selecting a Row in a Spreadsheet

In the previous section it has been mentioned that selecting a row in a spreadsheet view is equivalent to the left button click event over a point in a scatter plot. Thus, if the user wants something to happen in response to selecting a row in a spreadsheet, then the user needs to set a callback function for the left button click event.

For coloring a point, selecting a row colors all points that correspond to that row's data while unselecting a row does not make any changes to the color of those points. Thus, coloring a point

is a property that cannot be toggled on and off. In contrast, highlighting and hiding points are both properties that can be toggled. Thus, if the response to the left button click is to highlight a point, then selecting a row highlights all points that correspond to that row's data and unselecting a row un-highlights all points that correspond to that row's data. Similarly, if the response to the left button click is to hide a point, then selecting a row hides all points that correspond to that row's data and unselecting a row makes those points reappear.

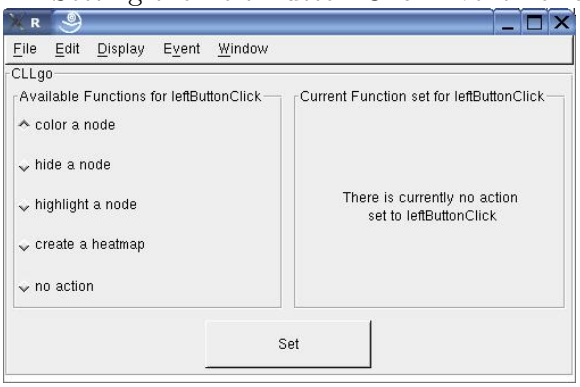
10.1.3 Left Button Click Event for a Graph

As mentioned previously, the currently available responses for a left button click event when the active `MVC` object's model is a graph are color a node, hide a node, highlight a node, create a heatmap, or no action.

Using the GUI

To see the callback functions that can be set to the left button click event when the active `MVC` contains a graph, first set the active `MVC` to CLLgo and then highlight the Event menu and select the Set Left Button Click menu item. Alternatively, the user can press Ctrl-B to activate the Set Left Button Click menu item. The control window now looks as shown in Figure 21.

Figure 21: Setting the Left Button Click Event For a Graph



Using the Command Line

To see the callback functions that can be set to the left button click event, call the function, getDescForEvent, with a parameter of "leftButtonClick". This function returns the descriptions of all possible callback functions for a certain event. It is this character string (the description) that is used as the shortName parameter in the setCallFunc function.

```
> setActiveMVC("CLLgo")
> getDescForEvent("leftButtonClick")
```

Coloring a Node

We will look at each callback function one at a time. First, let's set the left button click event to color a node.

Using the GUI

To do this using the GUI, choose the "color a node" radio button and click the Set button (after choosing the Set Left Button Click menu item under the Event menu). A new window appears that has a color browser to allow the user to choose a color that is used to color the nodes. Pick a color on the color wheel and click the Ok button. The window with the color wheel is shown in Figure 17.

Using the Command Line

To set the left button click event to color a node using the command line, call the function, setCallFunc, with "leftButtonClick" as the first parameter and "color a node" as the second parameter. The default value for the color is black. If the user wants a different color, then call the function, setColor. To see the current color, call the function, getColor.

```
> setActiveMVC("CLLgo")
> setCallFunc("leftButtonClick", "color a node")
> setColor("green")
```

Coloring a Node

Now click with the left button on a node in a plot of CLLgo. Note that this plot was created in Section 8.1. The node that was just clicked is filled in with the new color (I have used green as shown in Figure 22).

If two plots of the graph have been created, then a node in each plot is colored green, indicating that these are the same node. To keep the plots synchronized, some plotting information is stored with the data.

To choose a different color using the GUI, the user needs to again choose the "color a node" radio button and click the Set button, which causes the window with the color wheel to reappear. To choose a different color using the command line, call the function, setColor.

If a spreadsheet view of the node attributes is open, then the row in the spreadsheet that corresponds with the node that was just colored will be selected. You can also color a node by selecting a row in the spreadsheet. Please see Section 10.1.2 for how selecting a row in the spreadsheet is linked to clicking the left button over a plot.

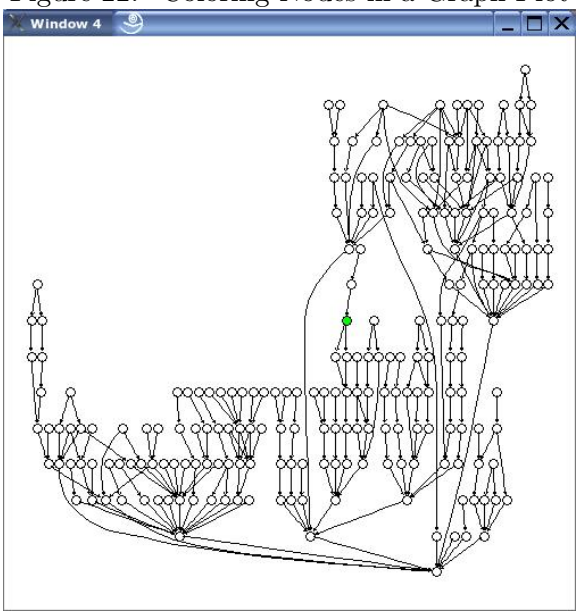
Hiding a Node

Next, let's set the left button click event to hide a node.

Using the GUI

To do this choose the "hide a node" radio button and click the Set button on the control window (after choosing the Set Left Button Click menu item under the Event menu).

Figure 22: Coloring Nodes in a Graph Plot



Using the Command Line

To set the left button click event to hide a node using the command line, call the function, setCallFunc, with "leftButtonClick" as the first parameter and "hide a node" as the second parameter.

Hiding a Node

Now click with the left button on a node in the plot of CLLgo. The node that was just clicked disappears. To make the node reappear, click with the left button over the spot where the node used to be and the node reappears. Thus, hiding a node is a property that can be toggled on and off.

Again, if a spreadsheet view of the node attributes is open, then the row in the spreadsheet that corresponds with the node that was just hidden will be selected. You can also hide a node by selecting a row in the spreadsheet. By unselecting a row in the spreadsheet, that node will reappear. Please see Section 10.1.2 for how selecting a row in the spreadsheet is linked to clicking the left button over a plot.

To make hidden values reappear, users can also use the Show Hidden Values menu item under the Edit menu. This is discussed in Section 10.1.1 and shown in Figure 19.

Highlighting a Node

Next, let's set the left button click event to highlight a node.

Using the GUI

To do this choose the "highlight a node" radio button and click the Set button on the control window (after choosing the Set Left Button Click menu item under the Event menu).

Using the Command Line

To set the left button click event to highlight a node using the command line, call the function, setCallFunc, with "leftButtonClick" as the first parameter and "highlight a node" as the second parameter.

Highlighting a Node

Now click with the left button on a node in the plot of CLLgo. The node that was just clicked is highlighted (i.e. the node boundary is colored red). If there is more than one plot of CLLgo, then all plots show a node that is highlighted, indicating that these nodes are the same. Similar to hiding, highlighting a node is a property that can be toggled on and off. To turn off the highlighting, just click with the left button over the same node.

Figure 23 shows a node that has been highlighted on the plot of CLLgo.

Again, if a spreadsheet view of the node attributes is open, then the row in the spreadsheet that corresponds with the node that was just highlighted will be selected. You can also highlight a node by selecting a row in the spreadsheet. By unselecting a row in the spreadsheet, that node will be un-highlighted. Please see Section 10.1.2 for how selecting a row in the spreadsheet is linked to clicking the left button over a plot.

Creating a Heatmap

Using the GUI

To do this choose the "create a heatmap" radio button and click the Set button on the control window (after choosing the Set Left Button Click menu item under the Event menu).

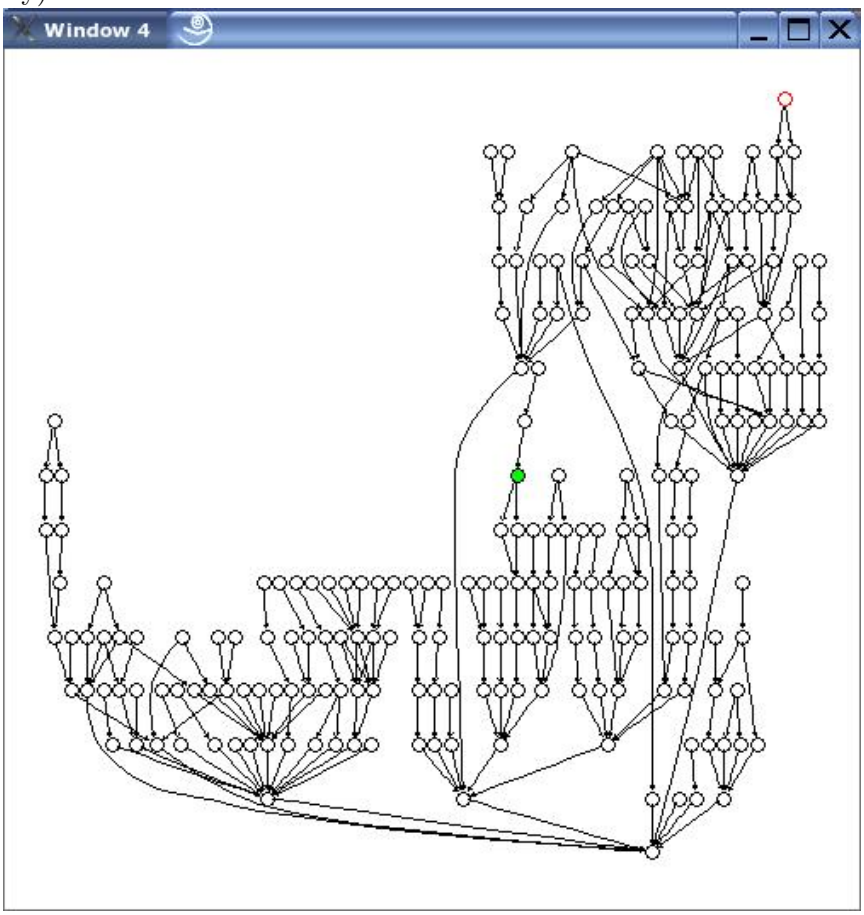
Using the Command Line

To set the left button click event to create a heatmap using the command line, call the function, setCallFunc, with "leftButtonClick" as the first parameter and "create a heatmap" as the second parameter.

Creating a Heatmap

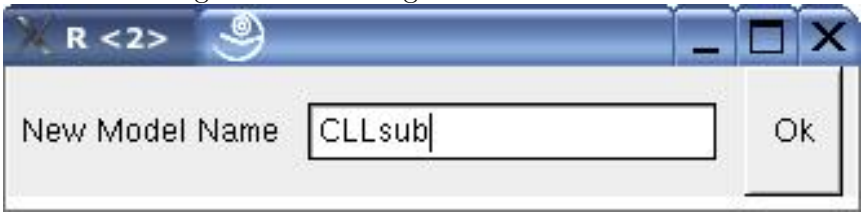
This is the most complex response to a left button click event. In this response, a new ExpressionSet model is created (that is a child of the current model) and a heatmap view of that data is created. By clicking the left button over a node, all genes that are annotated at that node will be included in the new child ExpressionSet model, which will contain the expression levels of these genes. Thus, an ancestor model of the graph must contain gene expression level data (i.e. an ExpressionSet model). Here, the CLLgo graph was created from the CLL model, which contains gene expression levels from CLL patients. Thus, this response is valid for the CLLgo model (however, it is not a valid response for the testGraph model, which is not derived from gene expression data).

Figure 23: Highlighting Nodes in the Plot of a Graph. The top right node is highlighted (has a red boundary).



Now click with the left button on a node in the plot of CLLgo. Any genes that are annotated at that node will be included in the child model and their gene expression levels will be shown in the new heatmap. Since you are creating a new model (and MVC), you must give a name to this new MVC. If the control window is open, a new window will appear asking for the name of the new model. This window is shown in Figure 24. As you can see, I named the new model, 'CLLsub'.

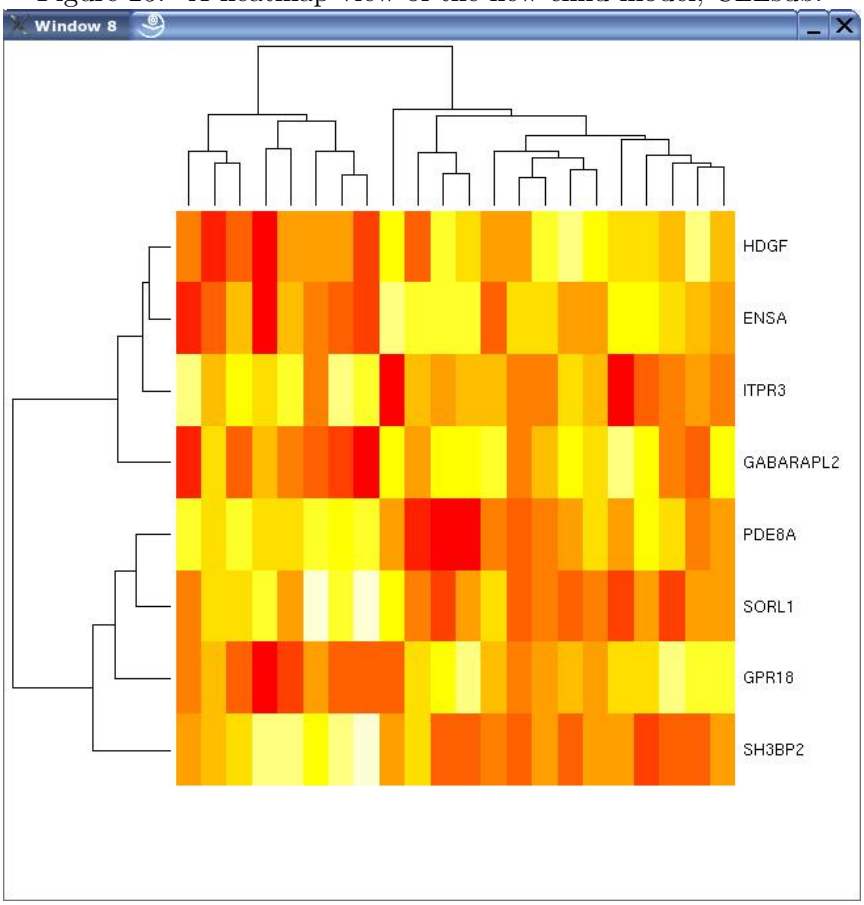
Figure 24: Naming the new child model.



If the control window is not open, then the user must give the name of the new model through the command line, using the function, setModelName. Before the user clicks with the left button over a node, the user must always call the setModelName function to set the name of the new child model. If this is not done, then a new child model and new heatmap view cannot be created. Calling the setModelName function must be done prior to each time the user clicks the left button over a node to create a new heatmap because each time a new model is created and it must be given a new name.

After clicking the left button over a node, a new window with a heatmap view appears. Recall that the gene expression levels shown in this heatmap correspond to the genes that are annotated at the node that was just clicked. In my example, the new heatmap view for node, signal transducer activity, is shown in Figure 25. This node (signal transducer activity) has 8 genes annotated at it and thus, the new heatmap view shows the expression levels for these 8 genes.

Figure 25: A heatmap view of the new child model, CLLsub.



Again, if a spreadsheet view of the node attributes is open, then the row in the spreadsheet that corresponds with the node that was just clicked by the left button will be selected. You can also create a heatmap by selecting a row in the spreadsheet. Please see Section 10.1.2 for

how selecting a row in the spreadsheet is linked to clicking the left button over a plot.

No Action

The last option and the default is to have no response to a left button click event. When the first view of the active data set is created, there is no response to a left button click event. If the user decides to color a node, hide a node, highlight a node, or create a heatmap in response to a left button click, the user can always reset the response to be nothing when a left button click happens.

Using the GUI

To do this choose the "no action" radio button and click the Set button on the control window (after choosing the Set Left Button Click menu item under the Event menu).

Using the Command Line

To set the left button click event to have no response using the command line, call the function, setCallFunc, with "leftButtonClick" as the first parameter and "" as the second parameter.

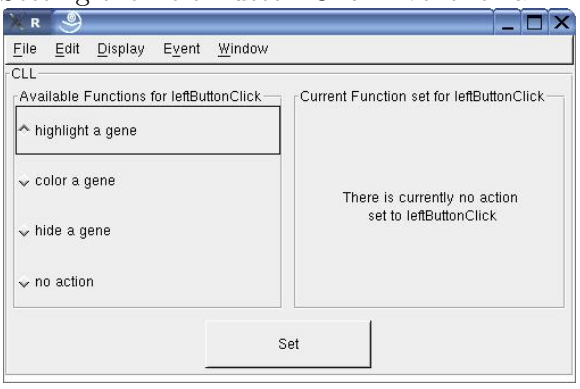
10.1.4 Left Button Click Event for an ExprSet

The currently available responses for a left button click event when the active `MVC` object's model is an ExpressionSet are color a gene, hide a gene, highlight a gene, or no action.

Using the GUI

To see the callback functions that can be set to the left button click event when the active `MVC` contains an ExpressionSet, first set the active `MVC` to CLL and then highlight the Event menu and select the Set Left Button Click menu item. Alternatively, the user can press Ctrl-B to activate the Set Left Button Click menu item. The control window now looks as shown in Figure 26.

Figure 26: Setting the Left Button Click Event for an ExpressionSet



Using the Command Line

To see the callback functions that can be set to the left button click event, call the function, getDescForEvent, with a parameter of "leftButtonClick". This function returns the descriptions of all possible callback functions for a certain event. It is this character string (the description) that is used as the shortName parameter in the setCallFunc function.

```
> setActiveMVC("CLL")
> getDescForEvent("leftButtonClick")
```

Coloring a Gene

We will look at each callback function one at a time. First, let's set the left button click event to color a gene.

Using the GUI

To do this using the GUI, choose the "color a gene" radio button and click the Set button (after choosing the Set Left Button Click menu item under the Event menu). A new window appears that has a color browser to allow the user to choose a color that is used to color the points. Pick a color on the color wheel and click the Ok button. The window with the color wheel is shown in Figure 17.

Using the Command Line

To set the left button click event to color a gene using the command line, call the function, setCallFunc, with "leftButtonClick" as the first parameter and "color a gene" as the second parameter. The default value for the color is black. If the user wants a different color, then call the function, setColor. To see the current color, call the function, getColor.

```
> setActiveMVC("CLL")
> setCallFunc("leftButtonClick", "color a gene")
> setColor("blue")
```

Coloring a Gene

Now click with the left button on a row in the heatmap of CLL. Note that this plot was created in Section 9.1.2. The row that was just clicked will now have a label that is colored blue. Also, users can click with the left button over the row dendrogram and all genes that are in that part of the dendrogram will have their labels colored blue. Figure 27 shows gene, TAGLN2, with a blue label.

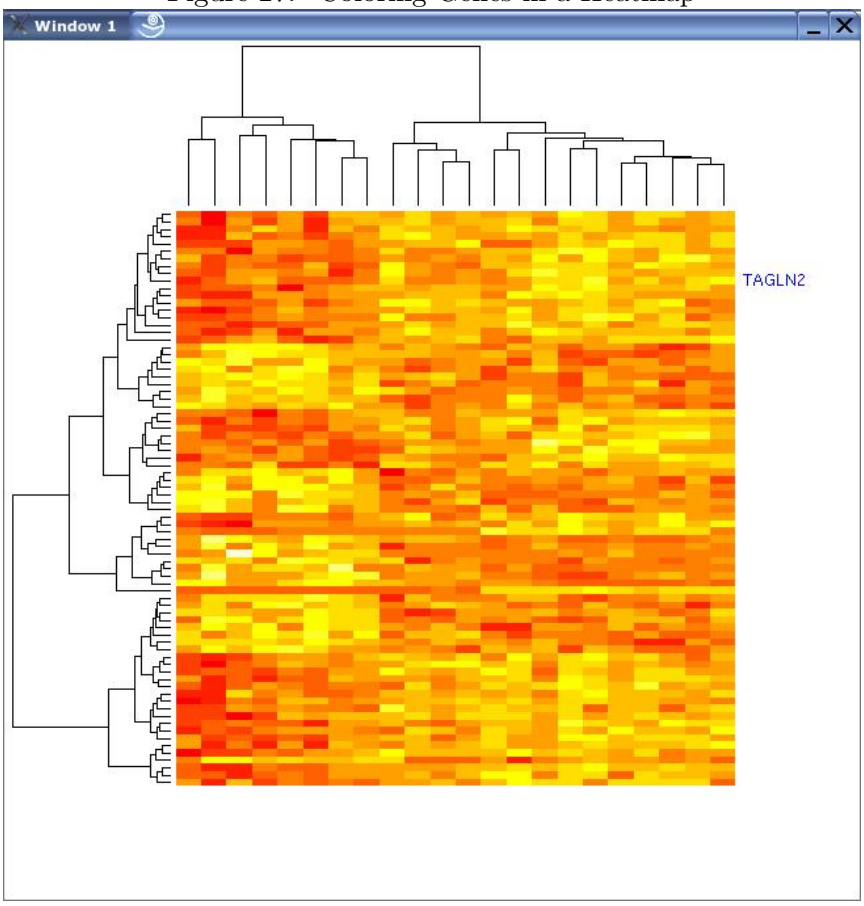
Hiding a Gene

Next, let's set the left button click event to hide a gene.

Using the GUI

To do this choose the "hide a gene" radio button and click the Set button on the control window (after choosing the Set Left Button Click menu item under the Event menu).

Figure 27: Coloring Genes in a Heatmap



Using the Command Line

To set the left button click event to hide a point using the command line, call the function, setCallFunc, with "leftButtonClick" as the first parameter and "hide a gene" as the second parameter.

Hiding a Gene

Now click with the left button on a row in the heatmap of CLL, which was created in Section 9.1.2. This will cause the entire heatmap to be redrawn so that this row is removed (also the row dendrogram will be recalculated since one gene is now missing). Thus, the response to hiding a gene is a bit slower because of the redraw rather than just updating a part of the plot.

The only way to reshow the hidden genes is through the Show Hidden Values menu item under the Edit menu. This option was discussed in Section 10.1.1 under the hiding a point section and is also discussed in Section 11.3. When the hidden genes are shown, the heatmap will again be completely redrawn and the row dendrogram will be recalculated.

Highlighting a Gene

Next, let's set the left button click event to highlight a gene.

Using the GUI

To do this choose the "highlight a gene" radio button and click the Set button on the control window (after choosing the Set Left Button Click menu item under the Event menu).

Using the Command Line

To set the left button click event to highlight a gene using the command line, call the function, setCallFunc, with "leftButtonClick" as the first parameter and "highlight a gene" as the second parameter.

Highlighting a Gene

Now click with the left button on a row in the CLL heatmap (or click with the left button over the row dendrogram). The row (representing a gene) that was just clicked is highlighted (i.e. a bold, italicized label with the gene name will appear next to the row). To turn off the highlighting, click on the highlighted row in the heatmap. Now the highlighting disappears. Thus, highlighting a gene is a property that can be toggled on and off.

Figure 28 shows a gene, AK2, that has been highlighted on the heatmap.

No Action

The last option and the default is to have no response to a left button click event. When the first view of the active data set is created, there is no response to a left button click event. If the user decides to color a gene, hide a gene, or highlight a gene in response to a left button click, the user can always reset the response to be nothing when a left button click happens.

Using the GUI

To do this choose the "no action" radio button and click the Set button on the control window (after choosing the Set Left Button Click menu item under the Event menu).

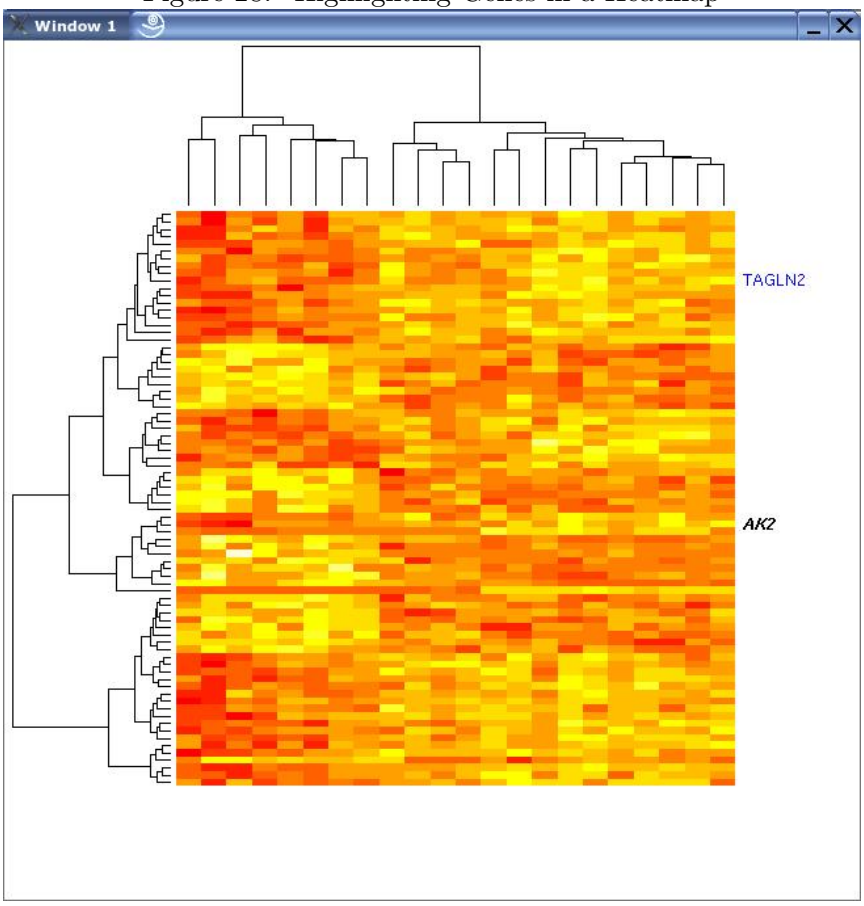
Using the Command Line

To set the left button click event to have no response using the command line, call the function, setCallFunc, with "leftButtonClick" as the first parameter and "" as the second parameter.

10.1.5 Left Button Click Event for a GSE (Gene Set Enrichment) Model

The currently available responses for a left button click event when the active `MVC` object's model is a GSE are color a point, hide a point, highlight a point, or no action. These responses are identical to the responses for the left button click event when the active `MVC` object's model is a data frame. Thus, please see Sections 10.1.1 and 10.1.2

Figure 28: Highlighting Genes in a Heatmap



10.2 Setting the Middle Button Click

The potential responses to a middle button click depend on the class of the active `MVC's` model. Thus, if the active `MVC's` model is of class `dfModel` (which means the model contains a data frame), then the potential responses to a middle button click are color a point, hide a point, highlight a point, or no action. If the active `MVC's` model is of class `graphModel` (which means the model contains a graph), then the potential responses to a middle button click are color a node, hide a node, highlight a node, create a heatmap, or no action. If the active `MVC's` model is of class `exprModel` (which means the model contains an ExpressionSet), then the potential responses to a middle button click are color a gene, hide a gene, highlight a gene, or no action. If the active `MVC's` model is of class `gseModel` (which means the model contains a GSE), then the potential responses to a middle button click are color a point, hide a point, highlight a point, or no action.

10.2.1 Middle Button Click Event for a Data Frame

As with the left button click event, the user currently has four options for the callback function for a middle button click event. These four options are color a point, hide a point, highlight a point, or no action. Thus, when the user clicks the middle button on a top of a point, the point can either be colored, hidden, highlighted, or nothing happens. The default is for nothing to happen. These callback functions have been described in Section 10.1.1 so please reference that section for more information on the callback functions.

The major difference between the left button click event and the middle button click event is the response to selecting or unselecting a row in a spreadsheet view. For the middle button click event, when a point is colored, hidden, or highlighted, the corresponding row in the spreadsheet view is selected. However, if a user directly selects a row in the spreadsheet, the callback function for a middle button click event does not occur. A user selecting a row in a spreadsheet always causes the left button click event to occur. When the user colors, highlights, or hides a point using the middle button click event, the corresponding row in the spreadsheet is selected only to keep the views synchronized. Similarly, unselecting a row does not cause the callback function for a middle button click event to be called. However, if a user removes the highlighting or hiding of a point through the middle button click event over a scatter plot, the corresponding row on the spreadsheet is unselected.

Note that the response to a right button click event behaves the same as the response to the middle button click event. Thus, if the user selects or unselects a row in a spreadsheet, the callback function for a right button click event does not occur.

Using the GUI

To find out what callback functions can be set to the middle button click event, highlight the Event menu and select the Set Middle Button Click menu item. Alternatively, the user can press Ctrl-M to activate the Set Middle Button Click menu item.

Using the Command Line

To see the callback functions that can be set to the middle button click event, call the function, getDescForEvent, with a parameter of "middleButtonClick". This function returns the descriptions of all possible callback functions for a certain event. It is this character string (the description) that is used as the shortName parameter in the setCallFunc function.

10.2.2 Middle Button Click Event for a Graph

As with the left button click event, the user currently has four options for the callback function for a middle button click event. These four options are color a node, hide a node, highlight a node, or no action. Thus, when the user clicks the middle button on a top of a node, the node can either be colored, hidden, highlighted, or nothing happens. The default is for nothing to happen. These callback functions have been described in Section 10.1.3 so please reference that section for more information on the callback functions. Also, please see Section 10.2.1

for information about how the node attribute spreadsheet responds to the middle button click event.

Using the GUI

To find out what callback functions can be set to the middle button click event, highlight the Event menu and select the Set Middle Button Click menu item. Alternatively, the user can press Ctrl-M to activate the Set Middle Button Click menu item.

Using the Command Line

To see the callback functions that can be set to the middle button click event, call the function, getDescForEvent, with a parameter of "middleButtonClick". This function returns the descriptions of all possible callback functions for a certain event. It is this character string (the description) that is used as the shortName parameter in the setCallFunc function.

10.2.3 Middle Button Click Event for an ExprSet

As with the left button click event, the user currently has four options for the callback function for a middle button click event. These four options are color a gene, hide a gene, highlight a gene, or no action. Thus, when the user clicks the middle button on a top of a row (representing a gene), the row can either be colored, hidden, highlighted, or nothing happens. The default is for nothing to happen. These callback functions have been described in Section 10.1.4 so please reference that section for more information on the callback functions.

Using the GUI

To find out what callback functions can be set to the middle button click event, highlight the Event menu and select the Set Middle Button Click menu item. Alternatively, the user can press Ctrl-M to activate the Set Middle Button Click menu item.

Using the Command Line

To see the callback functions that can be set to the middle button click event, call the function, getDescForEvent, with a parameter of "middleButtonClick". This function returns the descriptions of all possible callback functions for a certain event. It is this character string (the description) that is used as the shortName parameter in the setCallFunc function.

10.2.4 Middle Button Click Event for a GSE (Gene Set Enrichment) Model

As with the left button click event, the user currently has four options for the callback function for a middle button click event. These four options are color a point, hide a point, highlight a point, or no action. Thus, when the user clicks the middle button on a top of a point, the point can either be colored, hidden, highlighted, or nothing happens. The default is for nothing to happen. These callback functions have been described in Section 10.1.1 so please reference that section for more information on the callback functions. Also, please see Section 10.2.1 for

information about how the gene set spreadsheet responds to the middle button click event.

Using the GUI

To find out what callback functions can be set to the middle button click event, highlight the Event menu and select the Set Middle Button Click menu item. Alternatively, the user can press Ctrl-M to activate the Set Middle Button Click menu item.

Using the Command Line

To see the callback functions that can be set to the middle button click event, call the function, getDescForEvent, with a parameter of "middleButtonClick". This function returns the descriptions of all possible callback functions for a certain event. It is this character string (the description) that is used as the shortName parameter in the setCallFunc function.

10.3 Setting the Right Button Click

The potential responses to a right button click depend on the class of the active `MVC's` model. Thus, if the active `MVC's` model is of class `dfModel` (which means the model contains a data frame), then the potential responses to a right button click are color a point, hide a point, highlight a point, or no action. If the active `MVC's` model is of class `graphModel` (which means the model contains a graph), then the potential responses to a right button click are color a node, hide a node, highlight a node, create a heatmp, or no action. If the active `MVC's` model is of class `exprModel` (which means the model contains an ExpressionSet), then the potential responses to a right button click are color a gene, hide a gene, highlight a gene, or no action. Finally, if the active `MVC's` model is of class `GSE` (for gene set enrichment data), then the potential responses to a right button click are color a point, hide a point, highlight a point, or no action. For more information on these potential responses to a right button click, please see Sections 10.1 and 10.2.

10.4 Setting the Mouse Over Event

As with all the other events discussed in this section, the potential responses to a mouse over event depend on the class of the active `MVC's` model. Thus, if the active `MVC's` model is of class `dfModel` (which means the model contains a data frame), then the potential responses to a mouse over event are show tooltips over points, highlight points as cursor moves over them, or no action. If the active `MVC's` model is of class `graphModel` (which means the model contains a graph), then the potential responses to a mouse over event are show tooltips over nodes or no action. If the active `MVC's` model is of class `exprModel` (which means the model contains an ExpressionSet), then there are currently no available responses to a mouse over event. If the active `MVC's` model is of class `GSE` (for gene set enrichment data), then the potential responses to a mouse over event are show tooltips over points, highlight points as cursor moves over them, or no action.

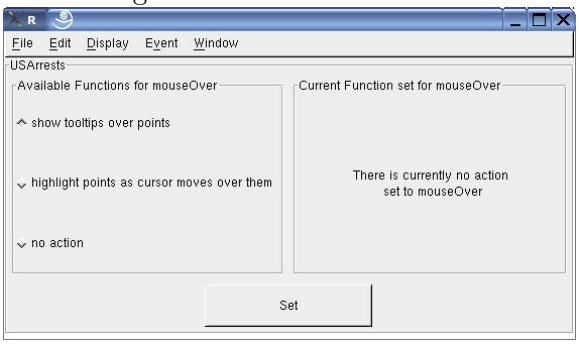
10.4.1 Mouse Over Event for a Data Frame

The currently available responses for a mouse over event when the active `MVC` object's model is a data frame are show tooltips over points, highlight points as cursor moves over them, or no action. The default is for nothing to happen.

Using the GUI

To see the callback functions that can be set to the mouse over event when the active `MVC` contains a data frame, first set the active `MVC` to USArrests and then highlight the Event menu and select the Set Mouse Over menu item. Alternatively, the user can press Ctrl-S to activate the Set Mouse Over menu item. The control window now looks as shown in Figure 29.

Figure 29: Setting the Mouse Over Event for a Data Frame



Using the Command Line

To see the callback functions that can be set to the mouse over event, call the function, getDescForEvent, with a parameter of "mouseOver". This function returns the descriptions of all possible callback functions for a certain event. It is this character string (the description) that is used as the shortName parameter in the setCallFunc function.

```
> setActiveMVC("USArrests")
> getDescForEvent("mouseOver")
```

Showing Tooltips over Points

We will look at each callback function one at a time. First, let's set the mouse over event to cause tooltips to appear over a point.

Using the GUI

To do this choose the "show tooltips over points" radio button and click the Set button (after choosing the Set Mouse Over menu item from the Event menu). A new window appears that allows the user to choose what tooltips appear over the points. The user is able to choose either the row names or any of the variables (column names) from the data frame as the tooltips. The window that appears for the "USArrests" data frame is shown in Figure 30.

Figure 30: Tooltip Options

For now, choose the "Row Names" radio button and click the Set Tooltips button.

Using the Command Line

To set the mouse over event to show tooltips using the command line, call the function, setCallFunc, with "mouseOver" as the first parameter and "show tooltips over points" as the second parameter. The default value for the tooltips is to show the row names of the data frame. If the user wants to use different values for the tooltips, then call the function, setTooltips. To see the current values for the tooltips, call the function, getTooltips.

```
> setActiveMVC("USArrests")
> setCallFunc("mouseOver", "show tooltips over points")
> setTooltips("rowNames")
```

Showing Tooltips

Now go to one of the scatter plots and move the cursor over the points (make sure the device is active by clicking on the plot first). A tooltip appears with the row name of that point in the tooltip window. Note that this is the one callback function where the views are not linked. Tooltips appear over the points in the active plot, but nothing happens in the other plots. To see linked plots with the mouse over event, please see the information on highlighting points as the cursor moves over them.

Figure 31 shows a tooltip (for row "Arizona") over a point on the scatter plot.

Highlighting Points as the Cursor Moves Over Them

Next, let's set the mouse over event to highlight a point.

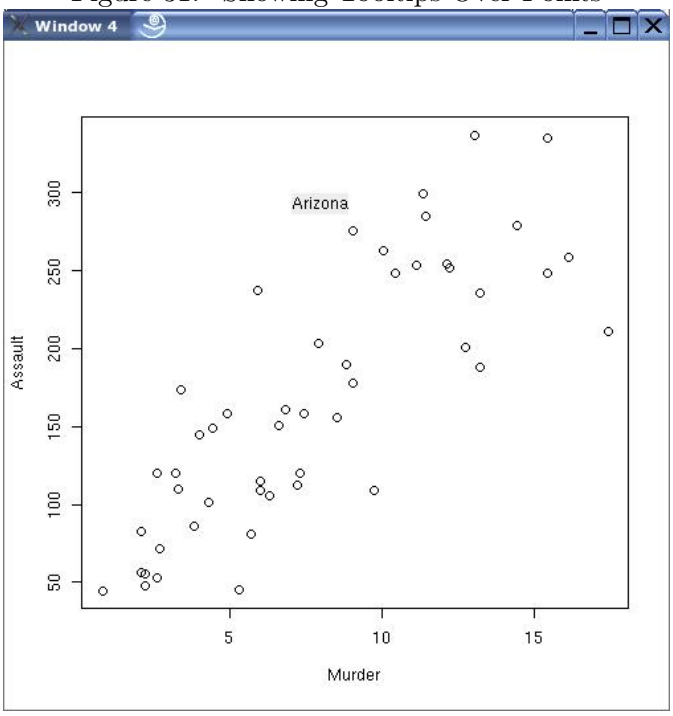
Using the GUI

To do this choose the "highlight points as cursor moves over them" radio button and click the Set button on the control window (after choosing the Set Mouse Over menu item from the Event menu).

Using the Command Line

To set the mouse over event to highlight a point using the command line, call the function, setCallFunc, with "mouseOver" as the first parameter and "highlight points as cursor moves over them" as the second parameter.

Figure 31: Showing Tooltips Over Points



Highlighting Points

Now move the cursor over the points in a scatter plot. As the cursor lingers over a point, that point is highlighted and then when the cursor moves off the point, that point is un-highlighted. Unlike the tooltips option for the mouse over event, all plots are linked so any corresponding points (i.e. points that show data from the same row in the data frame) are also highlighted in other plots. Also if the spreadsheet view is open, when a point is highlighted, then the corresponding row in the spreadsheet is selected and when the point is un-highlighted, the corresponding row in the spreadsheet is unselected.

No Action

The last option and the default is to have no response to a mouse over event. When the first view of the active data set is created, no response occurs when the cursor moves over the plot. If the user decides to show tooltips or highlighting over points in response to a mouse over event, the user can always reset the response to be nothing for the mouse over event.

Using the GUI

To do this choose the "no action" radio button and click the Set button on the control window (after choosing the Set Mouse Over menu item under the Event menu).

Using the Command Line

To set the mouse over event to have no response using the command line, call the function, setCallFunc, with "mouseOver" as the first parameter and "" as the second parameter.

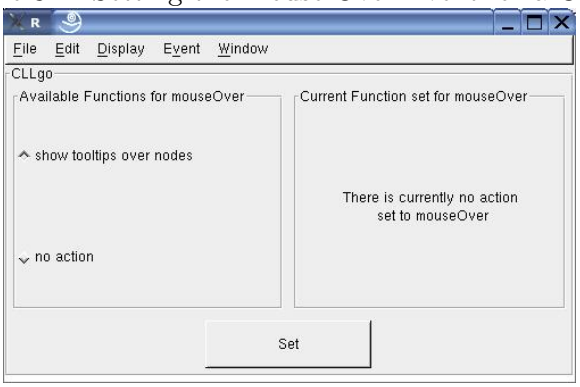
10.4.2 Mouse Over Event for a Graph

The currently available responses for a mouse over event when the active `MVC` object's model is a graph are show tooltips over nodes, or no action. The default is for nothing to happen.

Using the GUI

To see the callback functions that can be set to the mouse over event when the active `MVC` contains a graph, first set the active `MVC` to CLLgo and then highlight the Event menu and select the Set Mouse Over menu item. Alternatively, the user can press Ctrl-S to activate the Set Mouse Over menu item. The control window now looks as shown in Figure 32.

Figure 32: Setting the Mouse Over Event for a Graph



Using the Command Line

To see the callback functions that can be set to the mouse over event, call the function, getDescForEvent, with a parameter of "mouseOver". This function returns the descriptions of all possible callback functions for a certain event. It is this character string (the description) that is used as the shortName parameter in the setCallFunc function.

```
> setActiveMVC("CLLgo")
> getDescForEvent("mouseOver")
```

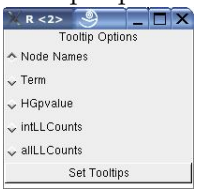
Showing Tooltips over Nodes

We will look at each callback function one at a time. First, let's set the mouse over event to cause tooltips to appear over a node.

Using the GUI

To do this choose the "show tooltips over nodes" radio button and click the Set button (after choosing the Set Mouse Over menu item from the Event menu). A new window appears that allows the user to choose what tooltips appear over the points. Currently, the user can choose the node names or any of the node attributes as the tooltips (if a graph has no node attributes, then the user can only choose the node names). The window that appears for CLLgo is shown in Figure 33.

Figure 33: Tooltip Options for a Graph



Choose the "Node Names" radio button and click the Set Tooltips button.

Using the Command Line

To set the mouse over event to show tooltips using the command line, call the function, setCallFunc, with "mouseOver" as the first parameter and "show tooltips over nodes" as the second parameter. The default value for the tooltips is to show the node names of the graph. If the user wants to use different values for the tooltips (to a node attribute for example), then call the function, setTooltips. To see the possible node attribute names, please call the function, getNodeAttr. To see the current values for the tooltips, call the function, getTooltips.

```
> setActiveMVC("CLLgo")
> setCallFunc("mouseOver", "show tooltips over nodes")
> getTooltips()
```

Showing Tooltips

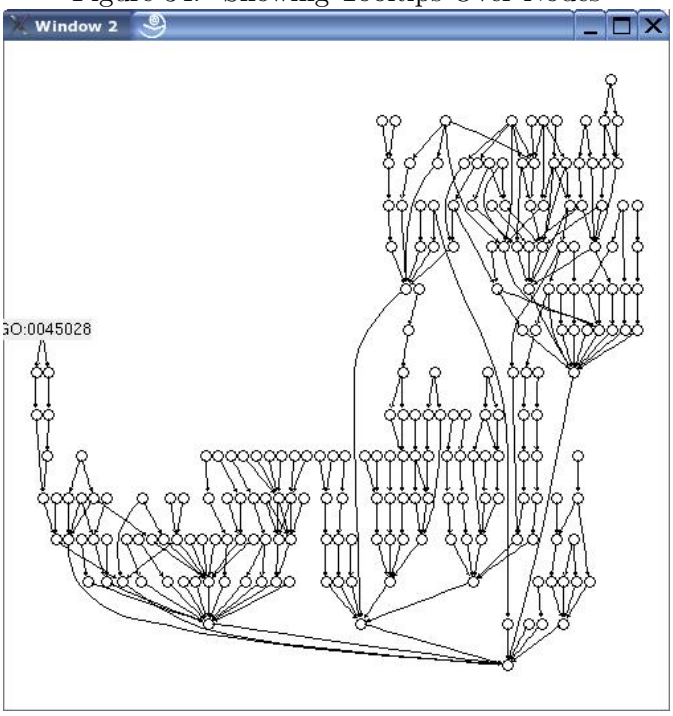
Now go to a plot of CLLgo and move the cursor over the nodes (make sure the device is active by clicking on the plot first). A tooltip appears with the node name of that node in the tooltip window. Note that this is the one callback function where the views are not linked. Tooltips appear over the nodes in the active plot, but nothing happens in the other plots.

Figure 34 shows a tooltip (for node "GO:0045028") over a node on the plot of CLLgo. In this instance, using 'Term' as the tooltips will be more helpful than the node names because the GO terms are more descriptive than the GO Ids (which are the node names).

No Action

The last option and the default is to have no response to a mouse over event. When the first view of the active data set is created, no response occurs when the cursor moves over the plot. If the user decides to show tooltips in response to a mouse over event, the user can always reset the response to be nothing for the mouse over event.

Figure 34: Showing Tooltips Over Nodes



Using the GUI

To do this choose the "no action" radio button and click the Set button on the control window (after choosing the Set Mouse Over menu item under the Event menu).

Using the Command Line

To set the mouse over event to have no response using the command line, call the function, setCallFunc, with "mouseOver" as the first parameter and "" as the second parameter.

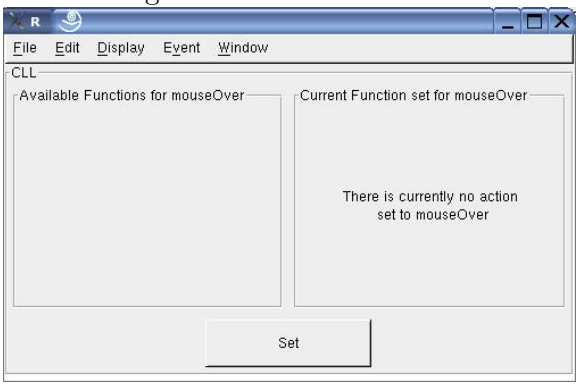
10.4.3 Mouse Over Event for an ExprSet

If the active `MVC` object's model contains an `ExpressionSet`, then there are no options for the response to a mouse over event. At this time, no callback functions have been created for this event.

Using the GUI

To see that there are no potential callback functions for the mouse over event when the active `MVC` contains an `ExpressionSet`, first set the active `MVC` to CLL and then highlight the Event menu and select the Set Mouse Over menu item. Alternatively, the user can press Ctrl-S to activate the Set Mouse Over menu item. The control window now looks as shown in Figure 35.

Figure 35: Setting the Mouse Over Event For an ExprSet



Using the Command Line

To see the callback functions that can be set to the mouse over event, call the function, getDescForEvent, with a parameter of "mouseOver". This function returns the descriptions of all possible callback functions for a certain event. It is this character string (the description) that is used as the shortName parameter in the setCallFunc function.

```
> setActiveMVC("CLL")
> getDescForEvent("mouseOver")
```

10.4.4 Mouse Over Event for a GSE (Gene Set Enrichment) Model

The currently available responses for a mouse over event when the active `MVC` object's model is a GSE (gene set enrichment data) are show tooltips over points, highlight points as cursor moves over them, or no action. The default is for nothing to happen.

All of these responses to a mouse over event have been described in Section 10.4.1 so please look at that section for more information on these responses. The only difference for a GSE is that the available tooltips are now the 'Gene Set Names', the 'gene set test statistics', and any model variables that have been created. The default tooltip values are to show the 'Gene Set Names'.

11 Editing Data

11.1 Deleting a `MVC` Object

If the user decides that one of the loaded `MVC` objects is no longer of interest, then the user has the option of deleting this `MVC` object. Only `MVC` objects that are not active can be deleted. Thus, if the user wants to delete a `MVC` object that is currently active, the user needs to set

another MVC object to be active first. Setting the active MVC object was discussed in Section 4.

If only one MVC object is loaded, then no MVC objects can be deleted. When there is only one MVC object that means the one MVC object must be the active MVC object and the active MVC object cannot be deleted.

Deleting a MVC object removes all aspects of the MVC object. In other words, the data are removed, all views of this data set are closed, and any information pertaining to this MVC object is deleted. This is not reversible. Note that if the user wants to delete all MVC objects, then the user should quit the program. See Section 12 for more information on quitting.

Linked data sets were mentioned in Section 1.1 and how they are created was discussed in Section 5. They are mentioned here because by deleting a MVC object, the user may affect data sets that are linked. The example given in Section 1.1 was microarray data that is linked to meta-data, such as a Gene Ontology (GO) graph. This linking shows a parent-child relationship between the two data sets. The parent data set is the microarray data and the child data set is the GO graph because the terms included in the GO graph are determined from the data in the microarray. Since there is a one-to-one relationship between a data set and a MVC object (as discussed in Section 4), that means there is a parent MVC and a child MVC. The parent MVC object contains the microarray data in its model and the child MVC object contains the GO graph in its model. Thus, if the MVC object containing the microarray data is deleted, then the MVC object containing the GO graph no longer has a parent MVC object. Users must be careful when deleting a MVC object that they are not deleting a relationship they are interested in studying.

Using the GUI

Select the Delete MVC menu item under the Edit menu or press Ctrl-E. If there is more than one MVC object loaded, then a new window appears with the names of the non-active MVC objects. Users can then choose one of the non-active MVC objects to delete by selecting the radio button next to the MVC object that they want to delete and clicking the Delete button.

Using the Command Line

To delete a non-active MVC object from the command line, call the function, deleteMVC. The only parameter that is required by this function is the name of the MVC object to delete. If the user is not aware of the names of the loaded MVC objects, the user can call getModelNames to see the names of the loaded MVC objects. If the user wants to see the active MVC object's name, the user can call getActiveMVC. Recall that the active MVC object cannot be deleted.

```
> loadData(USJudgeRatings, "USJudges", "data.frame")
> getModelNames()
> getActiveMVC()
> deleteMVC("USJudges")
> getModelNames()
```

11.2 Creating the Model Graph

To help the user quickly see the relationships between the loaded `MVC` objects, the user can create a graph of the loaded `MVC` objects. In this graph, the nodes represent loaded `MVC` objects and edges indicate linking between the models stored in the `MVC` objects. Any `MVC` objects that are related are linked by a directed edge with the edge starting at the node of the parent `MVC` object and ending at the node of the child `MVC` object. If there is no linking between any of the loaded `MVC` objects, then the graph has no edges. Creating linked `MVC` objects is discussed in Section 5.

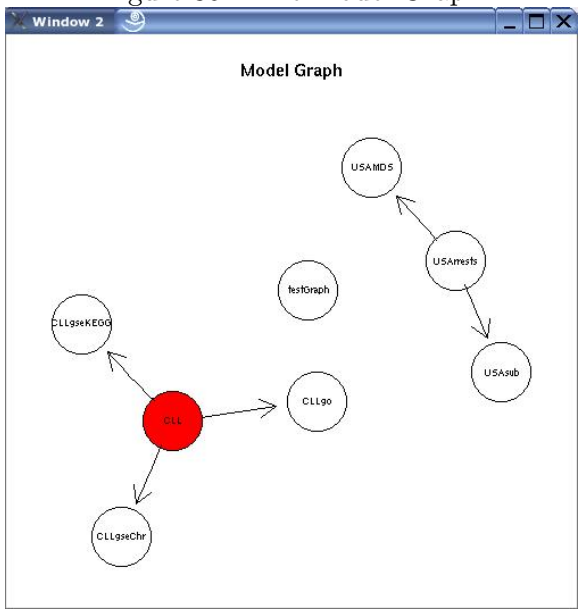
When this graph is plotted, the node that represents the active `MVC` object is colored red so that the user can quickly see which `MVC` is active. If the user changes the active `MVC` while the plot is open, the plot is updated to reflect the new active `MVC`. Also, if new `MVC` objects are loaded, the plot is updated with a new node to represent the new `MVC` object.

Currently, three `MVC` objects have been loaded in the code in Section 3 and five `MVC` objects have been created as child `MVCs` in the code in Section 5. Thus, there is linking between these objects as five of the `MVCs` are children of the three loaded `MVCs`.

Using the GUI

Select the Create Model Graph menu item under the Edit menu or press Ctrl-G. A new window opens that contains a plot of the model graph. The active `MVC` object is represented as a red colored node. This plot, which is shown in Figure 36, is not interactive.

Figure 36: The Model Graph



Using the Command Line

To create the model graph from the command line, there are two functions the user can call: createModelGraph and plotModelGraph. createModelGraph returns the model graph object, but does not plot the graph. plotModelGraph creates a plot of the model graph. Neither function requires any parameters. The only reason a user may want to call createModelGraph is if the user wants direct access to the graph object (for example, to use a different layout when plotting the graph).

```
> setActiveMVC("USArrests")
> x <- createModelGraph()
> plotModelGraph()
> setActiveMVC("testGraph")
```

11.3 Showing Hidden Values

If the user has hidden values for the active MVC, then these values can be reshown using the Show Hidden Values menu item under the Edit menu (or alternatively by typing Ctrl-H) or by using the showHiddenValues function. If the active MVC contains a data frame, then the hidden values will be row names; if the active MVC contains a graph, then the hidden values will be nodes; if the active MVC contains an ExpressionSet, then the hidden values will be genes; and if the active MVC contains a GSE, then the hidden values will be gene sets.

Using the Show Hidden Values menu item or the showHiddenValues function is particularly useful when the active MVC contains an ExpressionSet and it has hidden genes. The only way to reshow these hidden genes is through this functionality. Please see Section 10.1.4 for more information on hiding a gene.

Using the GUI

Select the Show Hidden Values menu item under the Edit menu or press Ctrl-H. If the active MVC has hidden values in its model, then a new window will appear that allows the user to select which hidden values to show. This new window was shown in Figure 19. If the active MVC has no hidden values, then nothing will happen when the Show Hidden Values menu item is selected.

Using the Command Line

To show hidden values from the command line, call the function, showHiddenValues. To see which values are hidden for the active MVC, call the function, getHiddenValues. Please see the man pages for these functions for more information.

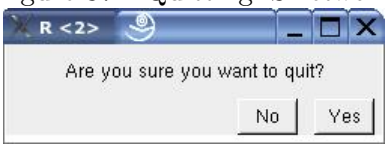
```
> setActiveMVC("CLL")
> hidVal <- getHiddenValues()
> showHiddenValues(hidVal)
```

12 Quitting

Using the GUI

When the user is ready to quit, select the Quit menu item under the File menu or press Ctrl-Q. A message box appears that asks if the user really wants to quit, as shown in Figure 37. By clicking the Yes button, all windows such as plots as well as the control window are closed. Also all data sets that have been loaded are removed so there are no loaded `MVCs` once the user has quit.

Figure 37: Quitting iSNetwork



Using the Command Line

To quit using the command line, call the function, quitiSNetwork. The quitiSNetwork function closes all views and removes all loaded data. This action is not reversible. Please see the man page for quitiSNetwork for more information.

13 Command Line Functions

Most of the command line functions have been described in the previous sections according to their functionality, but a quick review is given here. All of the operations that can be performed through the GUI can also be performed using the R command line. To find more information about any of the command line functions, please see their man pages.

The load data function is loadData. Please see Section 3.2 for more information.

If an ExpressionSet data set is loaded with an empty annotation slot, then the setChipType function sets this information, which is needed if the user wants to see gene symbols (rather than Affymetrix identifiers) on the heatmap view of the data.

To load model variables to an already loaded model, call the function, loadModelVar. To see the loaded model variables, call the function, getModelVar. Please see Section 3.3 for more information.

To see what the active `MVC` object is, the user can call getActiveMVC and to set the active `MVC` object, the user can call setActiveMVC. Please see Section 4.2 for more information.

To create views of the active `MVC's` model, there are many functions depending on the model in the active `MVC`. If the active `MVC` contains a data frame model, then there are two functions: createSpreadsheet, which creates a spreadsheet, and createSPlot, which creates a scatter plot.

Please see Sections 7.1.2 and 7.2.2 for more information. If the active MVC contains a graph model, then there are two functions: createGraphPlot, which creates a plot of the graph, and createSpreadsheet, which creates a spreadsheet of the node attributes. Please see Section 8.1.2 for more information. If the active MVC contains an ExpressionSet model, then there is one function to create a view: createHeatmap, which creates a heatmap of the expression data. Please see Section 9.1.2 for more information. If the active MVC contains a GSE model, then there are two functions to create views: createQQplot, which creates a qq-plot for the gene set statistics (using the qqnorm function), and createSpreadsheet, which creates a spreadsheet view of information on the gene sets.

There are several functions to help a user link a callback function to an event. The getEvents function returns the names of the events that can have callback functions linked to them (currently the events are leftButtonClick, middleButtonClick, rightButtonClick, and mouseOver). The getDescForEvent function returns all possible callback functions for an event (note that this function does not return the "no action" option, though that is an option). Finally, the setCallFunc actually links a callback function to an event. The setCallFunc function provides the same functionality as all of the menu items under the Event menu. Please see Section 10 for more information.

Note that a few callback functions require some extra information. These callback functions are color a point, color a node, color a gene and show tooltips. For coloring a point, coloring a node and coloring a gene, the users must say which color should be used and for showing tooltips, users must say what the tooltips should show (for example, row names, values from a column in the data, node names, node attributes, or model variables). To allow users to input this information, there are two command line functions: setColor and setTooltips. Also, if users want to know what these values are set to, they can call getColor or getTooltips. For a graph, the tooltips can be either the node names or a node attribute. To see the names of the node attributes, call the function, getNodeAttr. For more information on these callback functions, see Section 10.1.1 for coloring a point, see Section 10.1.3 for coloring a node, Section 10.1.4 for color a gene, and see Sections 10.4.1 and 10.4.2 for showing tooltips.

To delete a MVC object, call the function, deleteMVC. Please see Section 11.1 for more information.

To see how the loaded MVC objects are related, call the functions createModelGraph and plotModelGraph. The createModelGraph function creates a graph object where the nodes are the loaded MVC objects and the edges represent relationships between the MVCs. The edges start at the parent MVC and end at the child MVC. To see this graph object plotted, call the function plotModelGraph. Please see Section 11.2 for more information.

To see which values are hidden, call the function, getHiddenValues, and to show these hidden values, call the function, showHiddenValues. Please see Section 11.3 for more information.

To create linked MVC objects, there are several functions that let the user create child MVCs from already loaded MVCs. If the active MVC object contains a data frame model, then the functions to create a child MVC are createSubset and performMDS. Please see Section 5.1 for more information. If the active MVC contains an ExpressionSet model, then the functions

to create a child MVC are createGoGraph, createChrLocGSE, and createKEGGGSE. Please see Section 5.2 for more information. If the active MVC contains a graph model or a GSE model, there are currently no methods implemented to create a child MVC. To see what the available functions are to create a child MVC call the function, getMethodsToCreateChild, which returns the function names that can create a child MVC for the currently active MVC.

Another method to create a child model is to set the response to a click event on a graph to create a heatmap (see Section 10.1.3). This response creates a new child model and a heatmap view of this child model. This new child model must have a name that is unique from all the other loaded model names. If the user performs this interaction while using the GUI, then an interface will appear each time the user interacts with a node asking for the name of the new model. However, if the user is performing this interaction (to create a heatmap from a graph node) while using the command line, the user must set the model name before each interaction with a node, using the setModelName function.

To quit the program, call the function, quitiSNetwork. Please see Section 12 for more information.

To see the names of the loaded data sets (and thus, the names of the MVC objects), the user can call the function, getModelNames. This returns all the names of the currently loaded MVC objects.

To see all of the information about the loaded MVC objects (i.e. the model information, the view information, and the controller information), the user can call the function, getMVCList. This function returns a list of all the MVC objects that are currently loaded.

To see the model data for a particular MVC, the user can call the function getData.

To return the MVC object that corresponds with a view, the user can call the function, getMVCFromWinNum. This function is available for developers that want to create new callback functions for an event. In Section 14.2.2, getMVCFromWinNum is called within the new callback function, changePch. The getMVCFromWinNum function is needed by new callback functions because a user may interact with a view that is not from the active MVC object. Please see 10 for more information on how a user can interact with views that are not from the active MVC.

In Section 14 a few more command line functions, which let the user extend the *iSNetwork* package's functionality, are discussed. These functions include addCBFunction to add a callback function, addMenuItem to add a new menu item to the main menu, addSubMenuItem to add a new sub menu item, and addNewChildMethod to store a new method to create a child MVC object.

Also, there are a few functions that pertain to the GUI and these functions are createControlWindow, which creates the GUI, and resetWinSize, which resets the GUI to its default size.

For information on the accelerator keys that are currently in use, please call the function getKeyVals. To remove or add accelerator keys, the user has access to the functions, removeKeyVals and addToKeyVals, respectively.

A couple of functions are exported so that they are available to users of different software. The function, heatmapNI, is identical to the heatmap function, except now the row and column side images can be a matrix rather than a vector. The function, chrCats, explains how the chromosome categories in the MAP environment are converted into a list of categories. For example, the chromosome location '14q22' would be converted into the locations: 14, 14q, 14q2, and 14q22. Please see the man pages for these functions for more information.

A few functions are not meant to be called by the user. These include loadModel, setToggleX and setToggleY.

14 Extensibility in the *iSNetwork* Package

Having an extensible design was one of the goals for the *iSNetwork* package discussed in Section 1.1. Thus, there are several places where users can make additions based on the needs of their data. Some of the additions can be done directly at the R command line, such as adding new menus and menu items to the GUI, and adding potential callback functions for an event, while other additions are more complicated and require extending the code in the *iSNetwork* and/or the *MVCClass* package. Additions that require extending code in a package include creating new model or view classes and adding new events that views respond to. All of these possible ways of extending the functionality of the *iSNetwork* package are discussed in the following sections. All extensions of the *iSNetwork* package only require the user to program in R.

14.1 Adding New Model or View Classes

Currently in the *iSNetwork* package, the user has access to four types of models: data frame (or matrix), graph, ExpressionSet and GSE (gene set enrichment). For the data frame model, the user can create a spreadsheet view and a scatter plot view. For the graph model, the user can create a plot of the graph with different layout methods or node shapes and a spreadsheet of the node attributes. For the `ExpressionSet` model, the user can create a heatmap view. For the `GSE` model, the user can create a qq-plot for the gene set statistics or a spreadsheet view of gene set information. If the user wants additional model and view classes to be available, the user should start by reading the *MVCClass* and *BioMVCClass* Vignettes to see which model and view classes are currently defined in those packages. The *MVCClass* and *BioMVCClass* Vignettes also show the inheritance structure for the classes.

14.1.1 Adding a New Model Class

To add a new model class that the *iSNetwork* package could use, the user needs to create a new class definition and potentially generic function definitions that extend the classes in the *MVCClass* or the *BioMVCClass* package. After the new model class is defined, the user needs to make several extensions to the *iSNetwork* package. These extensions to the *iSNetwork* package include creating definitions for the initialize and the updateModel methods for the

new model class. Also, two pieces of code in the *iSNetwork* package need to be changed: the loadModel function needs to include the new model type and the GUI for the Load Model menu item needs to change to include this new type of model (this is set in the setLoadModelView function).

14.1.2 Adding a New View Class

Similarly to adding a new model class, adding a new view class requires the user to create a new class definition and potentially generic function definitions that extend the *MVCClass* and *BioMVCClass* packages. For example, if the user wants to create a view class for a histogram, then the user needs to create a class, which could be called `hPlotView`, and this new class inherits from the `plotView` class. See the *MVCClass* Vignette for more information on these view classes.

After the new view class is defined, the user needs to make several extensions to the *iSNetwork* package. The user needs to create definitions for the initialize, the updateView, and the redrawView methods. Depending on the events this new view class should respond to, the user may also need to create definitions for methods that respond to events, such as a clickEvent or a motionEvent method. The user also needs to add a menu item to the Display menu using the setDisplayMenu function, which is currently not exported from the *iSNetwork* package. The new Display menu item creates this new view when the menu item is activated. The necessary steps for adding a histogram view for the data frame model to the *iSNetwork* package are detailed in the Use Case below.

Use Case for Adding a Histogram

Primary Actor: Programmer
Scope: *iSNetwork* and *MVCClass* Package
Level: Summary
Stakeholders and Interests: Users of *iSNetwork*
Precondition: Programmer has access to the *iSNetwork* and *MVCClass* code
Success Guarantee: A new histogram view, which is an interactive and linked view, is available to for a data frame model.
Main Success Scenario:

1. Programmer adds a new submenu item to the Display menu on the control window by adding to the setDisplayMenu function in the *iSNetwork* package. This submenu item allows a user to create a histogram view when the active `MVC` contains a data frame model.
2. Programmer creates a function that is called when the submenu item (created in step one) is activated. This function allows the user to choose which variable to plot and then creates the histogram with the appropriate data from the active model.
3. Programmer creates a new view class that represents a histogram view that extends the classes in the *MVCClass* package. This new view class stores information, such as the column and rows that were used to create the histogram.

4. Programmer must define methods for the new histogram view class and these methods must include an initialize and an updateView method. The initialize method properly sets up the view, creates the view instance, and makes sure that the view can respond to certain events, such as key press, delete, focus in, and button clicks (see the next step). The updateView method updates the view when the model has changed. This method takes two parameters: object, which is the view object, and vData, which is a list of the following four elements: row name, column name, old value and new value. This information passed in the parameters allows the programmer to update the view so it is synchronized with the model.
5. For the histogram view to be interactive, the view needs to respond to the following events: key press event, delete event, focus in event, the mouse button press event, and potentially the mouse over event. The key press event adds accelerators to the view, the delete event ensures that the view list is current, and the focus in event ensures that the device is active when the histogram has the focus. The mouse button press event (to implement, need to create a clickEvent method) and the mouse over event (need to create a motionEvent method) allow the user to change plot information for the data (such as the color of a histogram bin) by interacting with the view. Similarly, the motion notify event, if the programmer wants to respond to it, would let the user interact with the view by moving the cursor over the view. Responding to these events is set up in the initialize method for this new view class, which is defined by the programmer.
6. Programmer creates a function that creates the histogram view and is meant to be called from the command line. This is necessary because users should be able to create views from both the GUI and the command line.

Extensions:

1. Other events besides those listed in step four may need to be noticed depending on what interactivity the programmer wants to support.
2. A controller, such as a slider, can be added to the histogram view to control the histogram bandwidth.

14.2 Adding Events and Callback Functions

14.2.1 Adding a New Event

Currently, callback functions can be linked to four events in the *iSNetwork* package. These events are the left button click event, the middle button click event, the right button click event, and the mouse over event. Other events are being responded to in the *iSNetwork* package, but the user is currently not allowed to change the callback function linked to these events. These events include the delete event, the focus in event, and the key press event. The key press event is slightly different in that by adding a menu item to the GUI, a new key press event is added. This is discussed in Section 14.3.

If users want to add a new event (like brushing) to this list, they need to perform the following steps.

1. Add this event to the possEvents variable in the environment, mvcEnv. This variable, possEvents, is used so that functions in the *iSNetwork* package know which events they must potentially respond to. This variable is defined in the setGlobalEnvVariables function. For example, if the new event was brushing, then "brushing" is added to the possEvents variable, which already has the values "mouseOver", "leftButtonClick", "middleButtonClick", and "rightButtonClick".
2. Add a new variable with the name of the event to the controller of each MVC object. For example, each controller has a mouseOver variable that stores information on the mouse over event for that `MVC`. If the new event is brushing, then the new variable in the controller is called brushing. This step is done by adding the new variable to the setControllerDefaults function.
3. Add a new menu item to the Event menu. When this menu item is activated, it calls the setCallFunEvent function with the name of the event.
4. Depending on the event, the user may need to add a new signal handler to catch this event in the initialize method of the views that respond to this event. Currently, all views have a signal handler to catch the button click events and the mouse over event.
5. The user also has to add a new method for this event. For example, the signal handler for the mouse over event on the scatter plot view calls the motionEvent method for the scatter plot view. A similar method needs to be defined for all views that respond to this new event. Note that by adding a new method, a new generic function must also be defined.

14.2.2 Adding a new callback function

To add a callback function for an event that is currently available, the user can call the function, addCBFunction, from the command line. The addCBFunction function can add potential callback functions for the following events: mouseOver, leftButtonClick, middleButtonClick, and rightButtonClick.

The addCBFunction function has five parameters, which are callFunction, shortName, preprocessingFun, eventLists, and typeModel. The callFunction parameter is the name of the callback function, the shortName parameter is a short description of what the callback function does, the preprocessingFun parameter is a vector of function names that must be called before the callback function, the eventLists parameter is a vector of event names that this callback function can be linked to, and the typeModel parameter is the types of models that this callback function can work with (for now the options are "graphModel", "dfModel", "exprModel", and/or "gseModel" because views of these models are interactive).

The callback function that is given in the callFunction parameter must take only one argument. That one argument must be one of the following: type character to represent a row name from

the data frame if the callback function acts on a MVC that contains a dfModel, type character to represent a node name from the graph if the callback function acts on a MVC that contains a graphModel, type character to represent a gene name from the ExpressionSet if the callback function acts on a MVC that contains an exprModel, or type character to represent a gene set name from the GSE if the callback function acts on a MVC that contains a gseModel. In other words, the one parameter of the callback function must be of a type that makes sense for the data stored in the model. The callback function must create and handle a gUpdateDataMessage so that the data is updated.

Not all callback functions need preprocessing functions so the preprocessingFun parameter can be set to NULL. If the callback function needs some information besides its one parameter, then the callback function must have one or more preprocessing functions. For example, when the callback function is to color a point, there must be a preprocessing function that has users set the color they want to use.

Currently, the eventLists parameter can be a vector containing one or more of the following events: "mouseOver", "leftButtonClick", "middleButtonClick", and "rightButtonClick". These are the events in the *iSNetwork* package that the user can change the response to.

Note that certain callback functions only make sense with views of certain types of models. For example, if the callback function colors a node, then the only type of model this works with is a graph so the typeModel parameter is set to "graphModel".

The following code adds a callback function that lets the user change the plotting character of a point. The new callback function, which is called changePch, is first defined and then a call to the addCBFunction function is made. In the call to addCBFunction, the callback function is "changePch", the description is "change the plotting character", the preprocessing functions are set to NULL, the events this callback function can be linked to are any of the button click events ("leftButtonClick", "middleButtonClick", and "rightButtonClick"), and the type of model is "dfModel". In this callback function example, the new plotting character has been set to a triangle by hard coding the new pch value in the code: newPchValue<-2. Most likely a preprocessing function should instead be used to let the user decide what the new plotting character is.

```
> changePch <- function(rowName) {
+     curMVC <- getMVCFromWinNum()
+     virtualData <- virtualData(model(curMVC))
+     colName <- "pch"
+     colIndex <- match(colName, colnames(virtualData))
+     rowIndex <- match(rowName, rownames(virtualData))
+     newPchValue <- 2
+     mData <- as.list(newPchValue)
+     names(mData) <- rowName
+     dfMessage <- new("gUpdateDataMessage", type = colName, mData = mData,
+         dataName = modelName(model(curMVC)), from = "")
+     handleMessage(dfMessage)
```

```
+ }
> addCBFunction(changePch, "change the plotting character", NULL,
+     c("leftButtonClick", "middleButtonClick", "rightButtonClick"),
+     typeModel = "dfModel")
> setActiveMVC("USArrests")
> getDescForEvent("leftButtonClick")
```

Now this callback function can be linked to the left button click event, for example, by using the Set Left Button Click menu item under the Event menu on the GUI or by calling the function, setCallFunc. Once the changePch function has been linked to the left button click event, then the user can left click on a point in a scatter plot and the point is redrawn as a triangle.

14.3 Adding Menu Items

If the user wants to add menu items to the main menu on the control window from the R command line, then the following two R functions allow the user to do that: addMenuItem and addSubMenuItem.

14.3.1 Adding Menu Items

The function addMenuItem adds a menu item to the menu bar on the control window. Thus, the new menu item appears after Window on the menu bar (the menu items on the menu bar are File, Edit, Display, Event, and Window).

To add an accelerator to this menu item, place an underscore before the letter that you want to be the accelerator. In the following code example, the accelerator is 'N' because there is an underscore placed before the 'N' in New.

The default modifier type for adding a menu item (parameter modType) is the Alt button. Thus, for the new menu item that the following code creates, the accelerator is Alt-N. See the man page for addMenuItem for descriptions of the function's parameters.

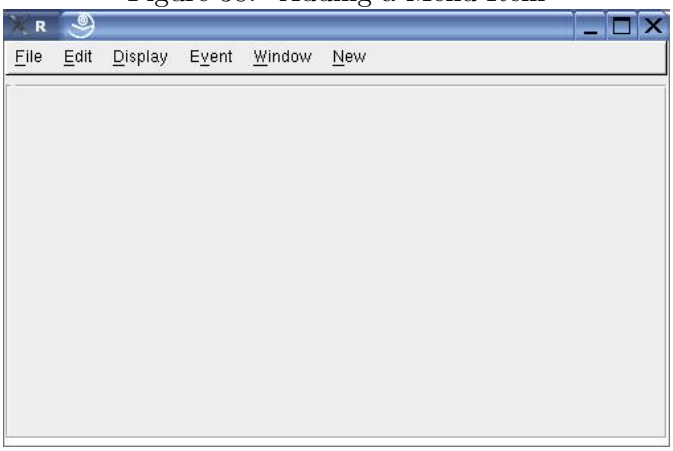
```
> addMenuItem(menuName = "new", labelText = "_New")
```

After performing the above code, the control window looks as shown in Figure 38.

14.3.2 Adding Sub Menu Items

If instead the user wants to add a sub menu item to an already existing menu, then the user should call the function addSubMenuItem. Examples of sub menu items that are already in the main menu are Load Model, Create Child Model, and Quit under the File menu.

Figure 38: Adding a Menu Item



Again, to add an accelerator to the sub menu item, place an underscore before the letter that you want to be the accelerator. In the following code example, the accelerator is 'N' because the underscore appears before the letter 'N' in Different.

The default modifier type for adding a sub menu item (parameter modType) is the Ctrl button. Thus, for the new sub menu item that the following code creates, the accelerator is Ctrl-N. See the man page for addSubMenuItem for descriptions of the function's parameters.

Also, the user must give a character string for action (the third parameter) that corresponds to the name of a function that is called when this sub menu item is activated. This function may have parameters. In the code below, action is set to "testfun", the name of the function that is defined in the following code chunk.

In the following example, the sub menu item, Different, is added to the menu item, New.

```
> testfun <- function() {
+     w <- gtkWindow(show = FALSE)
+     lab <- gtkLabel("It works!")
+     w$Add(lab)
+     w$Show()
+ }
> addSubMenuItem(menuName = "new", labelText = "Differe_nt  Ctrl+N",
+     action = "testfun")
```

If the user now chooses the Different menu item from the New menu, then a new window opens that says "It works!".

14.4 Adding Methods to Create Child MVCs

Adding a new method to create a child MVC means that at least three functions must be created. The first function creates the new child MVC object and the other two functions must link this new MVC object to its parent. Two functions are needed for linking because one function must link parent MVC to the child MVC and the second function must link child MVC to the parent MVC. More details on each of these three functions are given in the following sections. Then an example is given at the end.

14.4.1 Creating the Function to Create a New MVC

The first function, which creates a new MVC, starts with the active MVC and performs some type of action to create a new model from the active model (examples include creating a subset, performing MDS, etc.). Then with this new model, this first function creates and handles a gAddChildMessage object to create the new MVC object. Information about objects of class gAddChildMessage can be found in the *MVCClass* Vignette. The gAddChildMessage class has three slots: dataName, which is the name of the new MVC; mData, which is a list that contains the model data and virtual data to fill slots in the model; and type, which is the type of model (currently, the options are 'ExpressionSet", "graph", "data.frame", or "GSE"). When a gAddChildMessage object is initialized, these three slots must be filled. To actually create the new MVC object, the gAddChildMessage object must be handled. This is performed by calling the handleMessage method with four parameters. The signature for the handleMessage method actually only expects one parameter: the gAddChildMessage object, but three more parameters are needed to correctly initialize the new MVC object. These three parameters are toParent, fromParent, and subsetParentData. toParent is the function that links from child MVC to parent MVC, fromParent is the function that links from parent MVC to child MVC, and subsetParentData tells which values from the parent model were used to create the child model. These parameters to the handleMessage method must be named toParent, fromParent, and subsetParentData. Please see the code in Section 14.4.4 for an example of a call to the handleMessage method. The next few sections give more information on the toParent, fromParent, and subsetParentData parameters.

14.4.2 Creating a Function to Link Child to Parent MVC

The toParent function takes an object of class gSendParentMessage as its one parameter. The gSendParentMessage object contains the update data message that was used on the child MVC's model. The purpose of the toParent function is to convert this update data message from the child into an update data message that works for the parent MVC. Thus, the toParent function decides which elements of the parent model correspond to the elements in the child model that just changed. Then the toParent function creates a gUpdateDataMessage object that tells the parent MVC how it should be updated. It is the newly created gUpdateDataMessage for the parent MVC that is the return value from the toParent function. In the example code in Section 14.4.4 the toParent function is linkToParentByNodeName.

14.4.3 Creating a Function to Link Parent to Child MVC

The fromParent function takes an object of class gSendChildMessage as its one parameter. The gSendChildMessage object has two slots: the parentUpdateDataMessage slot, which contains the update data message that was used on the parent MVC, and the childName slot, which contains the name of the child MVC object that needs to be updated. The childName slot is needed because a MVC object can have more than one child MVC.

The purpose of the fromParent function is to convert this update data message from the parent into an update data message for the child MVC object that is named in the childName slot. Thus, the fromParent function must decide which elements in the child model correspond to the elements in the parent model that just changed. Also recall that a child model may have been created without using all of the data from the parent model. Thus, this function also needs to take into account if the element that just changed in the parent was actually used in the creation of the child. If for example, node "a" changed in the parent, but node "a" was not used to create the child, then there is no need to create an update data message for the child. That is why one of the parameters to the handleMessage method for gAddChildMessage must be subsetParentData, as mentioned in Section 14.4.1. Thus, the fromParent function first determines if the element that changed in the parent was used to create the child and if it was, then the fromParent function must convert the parent element into the corresponding child element that should be updated. Finally, the fromParent function creates a gUpdateDataMessage object that tells the child MVC how it should be updated and it is the gUpdateDataMessage that is the return value from the fromParent function. If the element that changed in the parent model was not used in the creation of the child model, then the fromParent function returns NULL. In the example code in Section 14.4.4 the fromParent function is linkToChildByNodeName.

14.4.4 Example of Creating a Method to Create a Child MVC

The following is a method to create a subgraph child MVC from a graph parent MVC. First the two link functions are defined (these are called linkToParentByNodeName and linkToChildByNodeName). Then the function that actually creates the child MVC is defined and it is called createSubGraph.

Note that this method to create a child can only be used at the command line. The code shown below does not force this method to appear on the GUI. To have the function show on the GUI, the user has to change the variable, createChild, that is stored in the controller slot of the graph MVCs and the user needs to define a function that lets the user decide the node names for the subgraph through the GUI.

The function, addNewChildMethod, stores this new method to create a child MVC and ensures that it is an option that appears when the user calls getMethodsToCreateChild from the command line.

```
> linkToParentByNodeName <- function(updateParentMessage) {
+     curchildUpdateDataMessage <- childUpdateDataMessage(updateParentMessage)
```

```
+     childName <- dataName(curchildUpdateDataMessage)
+     allMVCs <- getMVCList()
+     allMVCNames <- unlist(lapply(allMVCs, function(x) {
+         x@model@modelName
+     }))
+     mvcIndex <- match(childName, allMVCNames)
+     childMVC <- allMVCs[[mvcIndex]]
+     parentName <- parentMVC(childMVC)
+     curtype <- type(curchildUpdateDataMessage)
+     data <- mData(curchildUpdateDataMessage)
+     parentMessage <- new("gUpdateDataMessage", type = curtype,
+         mData = data, dataName = parentName, from = childName)
+     return(parentMessage)
+ }
> linkToChildByNodeName <- function(updateChildMessage) {
+     curparentUpdateDataMessage <- parentUpdateDataMessage(updateChildMessage)
+     parentName <- dataName(curparentUpdateDataMessage)
+     curChildName <- childName(updateChildMessage)
+     allMVCs <- getMVCList()
+     allMVCNames <- unlist(lapply(allMVCs, function(x) {
+         x@model@modelName
+     }))
+     mvcIndex <- match(curChildName, allMVCNames)
+     curChildMVC <- allMVCs[[mvcIndex]]
+     childModelData <- modelData(model(curChildMVC))
+     childNodeNames <- nodes(childModelData)
+     curtype <- type(curparentUpdateDataMessage)
+     data <- mData(curparentUpdateDataMessage)
+     changeNodes <- names(data)
+     if (any(changeNodes %in% childNodeNames)) {
+         if (all(changeNodes %in% childNodeNames))
+             childMessage <- new("gUpdateDataMessage", type = curtype,
+                 mData = data, dataName = curChildName, from = parentName)
+         else {
+             curIndex <- which(changeNodes %in% childNodeNames)
+             childMessage <- new("gUpdateDataMessage", type = curtype,
+                 mData = data[curIndex], dataName = curChildName,
+                 from = parentName)
+         }
+         return(childMessage)
+     }
+     else return(NULL)
+ }
> createSubGraph <- function(newModelName, nodeNames) {
+     aMVC <- getActiveMVC()
```

```
+     if (aMVC != "") {
+         allMVCs <- getMVCList()
+         allMVCNames <- unlist(lapply(allMVCs, function(x) {
+             x@model@modelName
+         }))
+         mvcIndex <- match(aMVC, allMVCNames)
+         curMVC <- allMVCs[[mvcIndex]]
+         controlEnv <- controller(curMVC)
+         curModelData <- modelData(model(curMVC))
+         if (is(curModelData, "graph")) {
+             curData <- getData(aMVC)
+             if (any(nodeNames %in% nodes(curData))) {
+                 curIndex <- match(nodeNames, nodes(curData))
+                 curIndex <- curIndex[!is.na(curIndex)]
+                 nodesToUse <- nodeNames[nodeNames %in% nodes(curData)]
+                 subData <- subGraph(nodesToUse, curData)
+                 subVirData <- virtualData(model(curMVC))[curIndex,
+                   ]
+                 newAddChildMessage <- new("gAddChildMessage",
+                   dataName = newModelName, type = "graph", data = list(data = subData,
+                     virtualData = subVirData))
+                 handleMessage(newAddChildMessage, toParent = linkToParentByNodeName,
+                   fromParent = linkToChildByNodeName, subsetParentData = nodeNames)
+                 assign("childName", "", controlEnv)
+                 print(paste(newModelName, "data set has been loaded."))
+             }
+         }
+     }
+ }
> addNewChildMethod("createSubGraph", "Create Sub Graph", "graphModel")
> setActiveMVC("testGraph")
> getMethodsToCreateChild()
> createSubGraph("testGrSub", nodeNames = c("a", "b", "c", "e",
+     "g", "i"))
> setActiveMVC("testGrSub")
```

References

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, 1995.

Emden Gansner and Stephen North. An Open Graph Visualization System and its Applications to Software Engineering. *Software: Practice and Experience*, 30(11):1203–1233, 2000.

Aravind Subramanian, Pablo Tamayo, Vamsi K. Mootha, and et al. Gene set enrichment anal-

ysis: A knowledge-based approach for interpreting genome-wide expression profiles. *Proceedings of the National Academy of Sciences*, 102:15545–15550, 2005.

Deborah Swayne, Di Cook, Andreas Buja, and Duncan Temple Lang. *GGobi Manual*, February 2002.

Deborah Swayne, Andreas Buja, and Duncan Temple Lang. Exploratory Visual Analysis of Graphs in GGobi. In *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)*, Vienna, Austria, 2003.

Luke Tierney. *LISP-STAT: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics*. Wiley-Interscience, 1990.

Elizabeth Whalen and Robert Gentleman. *Generalizing the Model View Controller Paradigm*. PhD thesis, Harvard University, 2006a.

Elizabeth Whalen and Robert Gentleman. *Interactive Analysis and Visualization of Microarray Data*. PhD thesis, Harvard University, 2006b.