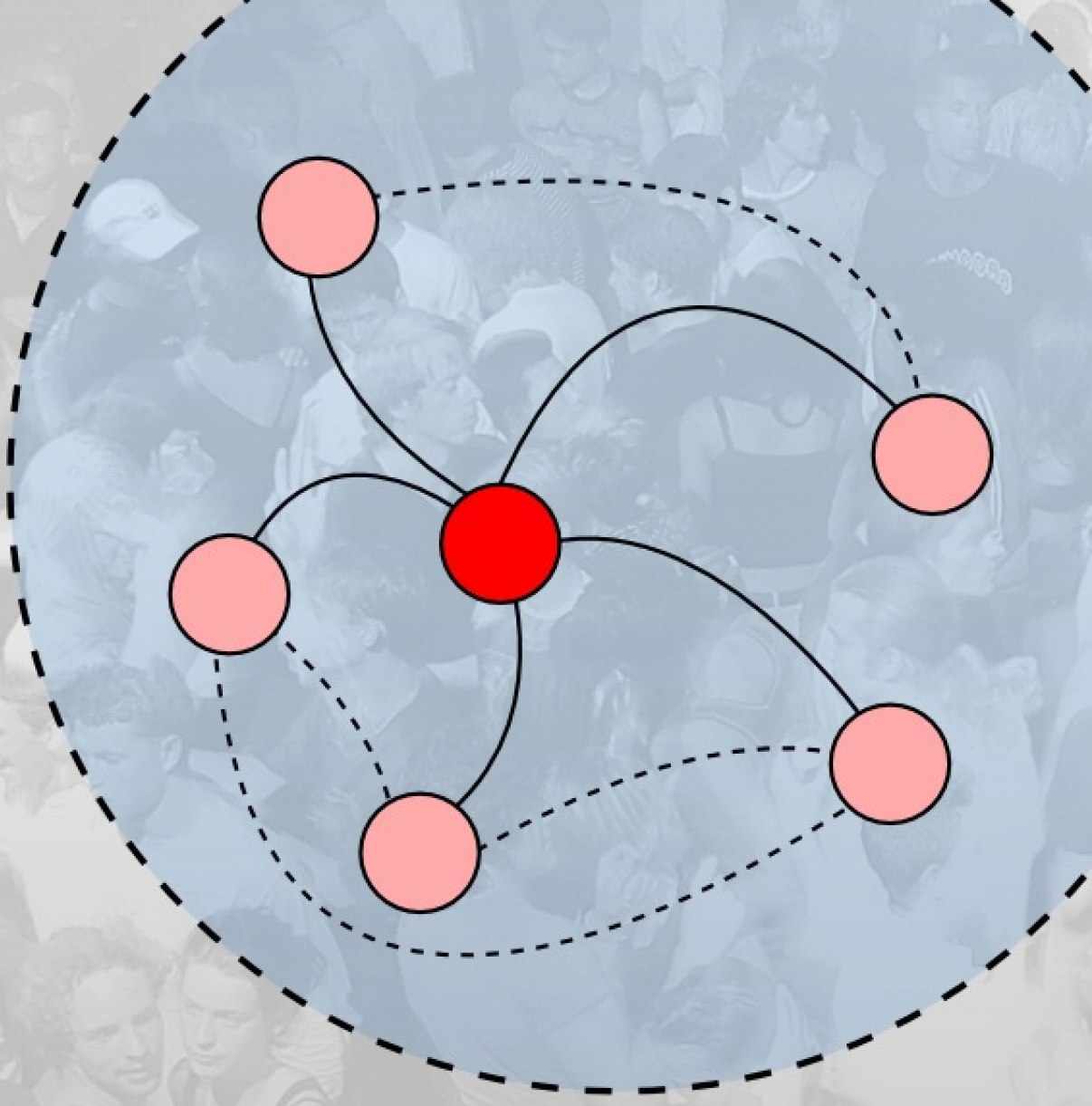


Egotistics

Version 0.9.0

Egotistics

Version 0.9.0

by Wojciech Gryc and Bernie Hogan

The software described in this manual was developed at NetLab, with Prof. Barry Wellman as Principal Investigator. It is currently unlicensed, and the creators have decided to retain rights to all the intellectual property within this software until it is publicly released, at which point it will be released under the GPL or a similar license. This guide is current as of 03:30 on December 27, 2007. For the latest version of the software, manual, and other accompanying tools and documents, please contact Wojciech Gryc at wojciech@gmail.com.

Table of Contents

About

Egotistics is software designed for two purposes: batch processing and the analysis of ego networks (personal networks). While a great deal of software exists for network analysis, few tools allow for the analysis of numerous small networks that have little or no relation to each other, or contain connected egos.

The software itself was designed in *Java 5.0*, and at the writing of this manual, is in the prototype phase. Specifically, the current major version is 0.5. The software itself is being tested and algorithms are being optimized. Any feedback would be appreciated as changes are made regularly.

The software itself was developed by Wojciech Gryc and Bernie Hogan at Dr. Barry Wellman's NetLab, at the University of Toronto, in Canada. The software is open source, but a license has yet to be chosen (it will be selected prior to the first stable, public release). Until then, please contact us prior to making modifications or redistributing the code.

To get in touch with us, feel free to e-mail Wojciech Gryc at wojciech@gmail.com.

System Requirements

Egotistics requires Python 2.2 or later to run, and Java 1.5.0.

Introduction

Egotistics is designed in a way to allow for easy batch processing of networks, and its code structure facilitates its extension with new algorithms and tools. For first-time users of Egotistics, there is a number of options available, both for network analysis and observing the data itself.

Creating Your First Project

When you first enter the software, it will not load any projects. There are a number of ways to start a project, though all assume that your data has already been collected. Note that Egotistics is not designed for data entry. Rather, it already assumes your data is stored within other files. Each project requires pre-existing networks, and the software will not function with such networks being included in a project file. It is recommended that networks be stored in tab-delimited formats in regular text. For example, a network with a 3-node clique would look like the following:

```
0	1	1
1	0	1
1	1	0
```

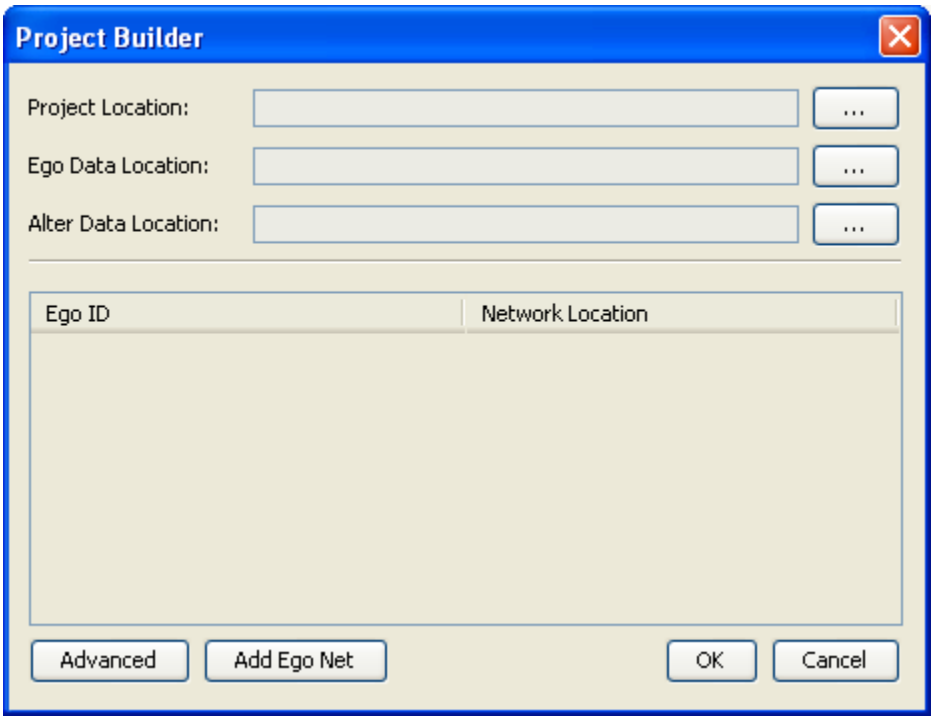
There are two ways to import basic networks. First, a new project must be started by going to *File* >

New Project. This will create an empty file with empty data. To add an individual network, you may then go to *Data* > *Import* > *Individual Network* and selected the proper network file there. While Egotistics assumes network files have a *.NET* extension, any file may be imported.

If you have multiple networks in one folder, going to *Data* > *Import* > *Network Folder* is a much more efficient option. Rather than choosing a network, you will now select a folder. All of the *.NET* files within this folder will then be imported into the project in much the same way as the individual networks were.

Using the Project Builder

Importing networks assumes that there is no data already attached to the network itself. If you have additional information for the networks you are importing, you may use the *Project Builder* utility instead. This tool facilitates both the importing of networks and data organized within *Comma Separated Values* (CSV) files. To launch the Project Builder, go to *Data* > *Import* > *Launch Project Builder*. You should see a window like the one below:



There are three key options for within the builder. First, you must select a *Project Location*. This is a file where all of the project information will be stored. The folder where the file is located will also contain all of the networks and external data.

The ego data is a file that contains ego-level data. There should only be one row for each network, with the first column being a unique "ego id" for the network. This column will not be editable in the software, and is very important. Ego IDs need not be numeric values.

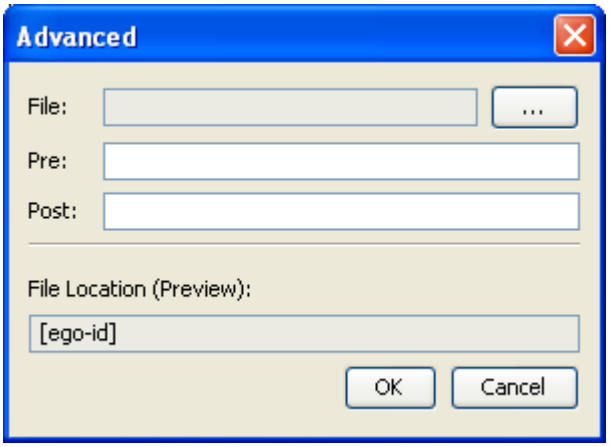
The alter data file contains alter-level information. There should be one row for each node in the network. The first two columns are extremely important in this file and also cannot be edited. The first column is the Ego ID, which corresponds to the ego network the node is a part of. The second column is the order. Because the network files are just matrices, the software must know which row in the alter-level data corresponds to the rows in the sociomatrices. This second column, the "order" does this. The order should start at *0* and end at *N-1*, where *N* is the number of nodes in the matrix. The CSV data should be organized in ascending order, according to the alter order.

Once you have added the two important CSV files, you can start adding networks. Click on *Add Network* and a new row will be added to the table. This has two columns, the first being the Ego ID that the network corresponds to, while the second is the network's location. To delete a network, left click on the row to make sure it's selected, and right click. This should bring up a *Delete Selected Row* option.

If your data does not contain alter-level or ego-level information, you may still use the tools above. Simply leave the text fields blank, and a new alter- or ego-level file will be created, with nothing but ego IDs and alter orders.

Adding Multiple Networks

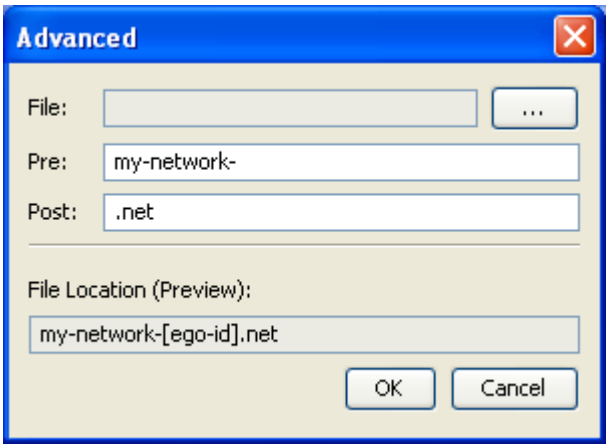
Adding individual networks is tedious, but does not have to be. If you have a large number of networks and would like to add them all at once, click the Advanced button in the Project Builder. This will bring up a new window that looks like the one below:



If you have a file (ego-level data, alter-level data, or plain text file) where the first line or first element in every row has the ego ID, select this file. The program will then find all of the unique ego Ids, and will import the networks from the same folder that file is in, using the name [ego-id]. Note that this does no assume a pre-existing extension on the file.

To further customize the importing process, you may type something in to the *Pre* and *Post* fields. This will then import all of the files that follow the naming scheme you design. For example, if every single network file in a folder is named "my-network-EGOID.net" where EGOID is the ego ID for each

network, you would write “my-network-” into the *Pre* field, “.net” into the *Post* field, and would see a preview of “my-network-[EGO-ID].net”.



Clicking “OK” will then add all of the ego IDs and network locations into the table within Project Builder.

Importing Data

As of version 0.5.2, *Egotistics* supports the importing of GraphML and Pajek file formats. Both options are available at *Data* > *Import*, followed by *GraphML* or *Pajek*, respectively. Please note that you using this import option will create a new project folder. Once you click on your choice, a window will pop up with three options, the *Source Folder*, *Project Folder*, and *Extension*. The first option allows you to choose the folder where the files are located, while the second option is where the new project will be created. Finally, you may also choose the extension of the files within the folder, as Pajek and GraphML files often come with different extensions.

Random Data

Egotistics also comes with the option of randomly generating data by selecting *Data* > *Random*. The submenu that pops up in this case allows for the random generation of Bernoulli graphs, allowing you to select the density of the graph and the number of alters. The density, in this case, represents the probability that an edge between two nodes will exist. The generated networks are undirected ones, and only one network is generated at a time.

Another random data generation features focuses on “categories”. Selecting this option allows one to categorize every alter within a network by a category labeled *1.0*, *2.0*, and so on. This data is automatically applied to every network in the project file, and the probability that an alter will be placed into any specific category is equal.

Recoding Data

Egotistics comes with the option of recoding data within projects. At this stage, this is only available through the console, but provides a very useful way to modify all ego-level, alter-level or network data at once. For example, if one is interested in symmetrizing all the graphs in a network and ensuring their edge weights are only equal to 1 or 0, one can use the code below:

```
# For an entire GraphProject.
Recode.symAndBin(CP)
# For a specific graph in a GraphProject.
x = CP.getEgoNet(3)
Recode.symAndBin(x)
```

To recode data in a specific column of a alter-level data, you can select a column and explain which values to recode. Note that the first column at the alter- and ego-level has an index of 0. It is recommended, however, that the first two columns of data in an *Egotistics* project not be recoded, as they contain important information for the software.

```
# For x from above.
Recode.data(x, 3, "old value", "new value")
```

Finally, advanced users have the option of a "merge" command, which allows multiple columns of data to be merged into one. This is useful when data imports are messy and need to be organized. Note that when merging, columns with a lower index have a higher precedence. For example, if you choose to merge columns 3 and 4, but both have data, then the data of column 3 will be retained, while the data in column 4 will be lost. Below are some examples:

```
# Merge two columns for every set of alters.
Recode.merge(CP, 3, 4)
# Merge all the columns with "_GENDER" into column 3.
# Note, however, that this is not case sensitive.
Recode.mergeByString(CP, "_GENDER", 3)
```

More information on advanced functions can be found in the *Egotistics JavaDocs*. The class that performs the recoding is under *edu.netlab.Recode*.

Exploring the Project

Egotistics comes with four main tabs which allow you to view the various aspects of the project you are working on. The first of these, and the one shown when the software loads, is the console. This is where code is executed and results of many calculations are displayed. Fortunately, a great deal of work has been put into the software to ensure one need not have any coding ability to use the software. However, if you want to create correlation matrices or see data that does not fit into the ego-level / alter-

level paradigm Egotistics employs, the output will be shown in the console. To learn more, please see the manual section *The Console*.

The next two tabs are called *Ego Attributes* and *Alter Attributes*. Simply put, these are tables that show the ego-level and alter-level data. The ego-level data table always shows information on all of the egos, while the alter table shows the attributes of all the alters of one ego at a time. When you select an algorithm from the *Ego Measures* or *Alter Measures* menus, most of the results will be appended to these two tables.

Finally, you have the *Network*, which shows the sociomatrix of the ego you are currently viewing.

To switch the view of the *Alter Attributes* and *Network* tabs, you can do so by selecting the proper ego id from the *Ego Net Selection* box.

Project Preferences

Under the Project Properties label at the side panel (shown above), two important properties may be set on the fly. Both of these modify the actual structure of the network being analyzed, but do not lead to permanent changes to the network.

`Append connected ego.`	Since ego networks sometimes contain a connected ego and sometimes do not, this option lets you add the ego when it is required. The network will receive a new node that is connected to every other node with a connection strength of 1. Note that measurements and calculations with this ego will not be added to either the ego-level or alter-level files, but may significantly affect the results of the other nodes.
`Remove isolates.`	If you would like to run your analysis while ignoring isolates, this option will allow you to do so. Any measurements will add an *N/A* label to the isolates themselves.
`Selecting both.`	If you select both options, then the network will first have its isolates removed, and then a connected ego will be added. The reason this order is used is that if you add a connected ego first, then no alter is an isolate.

The Console

Egotistics is a tool that is built in Java but implements *Jython* as a scripting environment. All of the functions available in the file menus and other graphical user interface (GUI) tools can also be accessed using the console.

Using the console is similar to most other scripting languages. For those familiar with the *Python* scripting language, *Jython* uses the same syntax, and comes with support for Java objects and classes.

The scripting language allows for simple mathematics, and interaction with classes already in the Egotistics software. Accessing methods for statistical analysis is simple, and often requires only passing one or more variable. The Egotistics *JavaDocs* show all the classes, and any methods that are labeled as static and public may be accessed using the console.

Indeed, the entire program can be used using the console, and the GUI itself can then be ignored. Below are some examples of using the software through the console:

```
g = LoadProject("E:\\myproject.dat")
```

Loading a project. Note that due to the way handles Strings, every backslash has to be represented as \\.

```
Components.countAll(g)
```

Counts the number of components in every ego network in the project and appends this information to the data.

```
ClusteringCoefficients.getClusterCoAllAlters(g)
```

Calculates the clustering coefficient of every alter in each network and appends the new data to every alter table in the project.

In addition to using methods and loading graphs or projects, the software also comes with a number of environment variables that dictate how the software itself works. While the GUI was designed to modify these without the console, the console also provides a way to access and modify them through code. These environment variables include:

`CP`	Standing for “Current Project”, CP is used by the software as a placeholder for the current project. If you load a project using the console, it is not accessible through the GUI unless you call it CP. Similarly, if you load a project using the GUI, it can be accessed using this name.
`UI`	The current instance of the program can be accessed through the UI identifier, and a number of common functions may be used to take advantage of the GUI. For example, UI.dispose() leaves the program, while UI.getFile() may be used to open the file dialog window and all you to retrieve a file name. See the JavaDoc for edu.netlab.ui.MainFrame for more information.
`Engine`	The Jython interpreter. This runs the actual scripting.
`NET#`	Reference to the current project's (if one is being shown using the graphical user inerface) networks, with # representing a number form 1 to *N*, where *N* is the number of networks in the project. This allows for easy access
`DATA#`	Same as above, but references each *GraphData* object in the current project.

Running Scripts Automatically

During the Egotistics startup, a script file called *autoexec.txt*, located in the program directory is run automatically. This file may be edited using code that would normally be typed into the console. Adding code to the window will force Egotistics to execute the new script whenever it starts. Keep in mind that removing any of the pre-existing code can disable features within the program. It is recommended that scripts be appended to the end of the file, and that pre-existing code remain unchanged.

Running from the Console / Terminal

It is possible to run Egotistics without using a graphical user interface. To do so, one can use the -s option within the software. To do this, use the DOS Prompt or Terminal Window and go to the Egotistics directory. Then type the following:

```
java –jar egotistics.jar –s <SCRIPT DIR>
```

In this case, change *<SCRIPT DIR>* to the location of the script file you are running. While the Egotistics console does not support loops, using this feature allows for creating for loops, while loops, conditional statements, and other similar structures. Standard Python syntax should be used when implementing such features.

Sample Script File

The script file below shows an example of how to create a new project, generate a five random networks, one category, calculate a few general network statistics, and then save the entire project to the original location.

```
NetWriter.newProject("/home/wojciech/Desktop/randomnets", "project.nxe")
P = GraphProject("/home/wojciech/Desktop/randomnets/project.nxe")
RandomGen.bernoulliNet(P,10,0.5)
RandomGen.bernoulliNet(P,30,0.4)
RandomGen.bernoulliNet(P,50,0.3)
RandomGen.bernoulliNet(P,70,0.2)
RandomGen.bernoulliNet(P,90,0.1)
RandomGen.alterCategory(P,2)
NetLabAssort.get(P,2)
Assortativity.r(P,2)
NetStats.numAlters(P)
NetStats.density(P)
NetWriter.saveProject(P)
```

Advanced Concepts

Egotistics was built for sociological research and works best with small networks. However, large networks are also supported, though in many cases, analysis may be slow.

Working With Numerous Large Networks

If your project contains a large number of networks, many of which have a few hundred or more

nodes, the amount of memory required by Egotistics will be extremely large. In most cases, the default heap size that Java allots for the software will be insufficient, and Egotistics will crash.

In order to run such a network, Egotistics should be launched from the command line, with custom settings for Java. Specifically, while Egotistics is normally launched at the command line with the following code,

```
java –jar egotistics.jar
```

it should now be launched with,

```
java –mx256m –jar egotistics.jar
```

where "256" is replaced by the number of megabytes of memory that will be allotted for use within the program. It is impossible to say how much memory should be allotted, as this is dependent on the project itself.

In cases where extremely large (over 1000 nodes) networks are used, it is advisable to write a script file and launch the software to automatically run the script without loading anything into the user interface (see *Running From Console / Terminal*). Most functions that accept a *GraphProject* object as a parameter will also accept a *GraphData* object as a parameter instead, thus saving computational time and memory. Combining output with the *Writer* class in *Egotistics*, which allows one to write directly to a file, can make processing large networks simple and quick. Below is an example where a set of *Pajek* files are loaded, turned into *GraphData* objects, and then analyzed.

```
w = Writer("/home/wojciech/Desktop/output.txt", 0)
# Supposing there are 10 networks, called net1.paj, net2.paj, etc.
for i in range(10):
        s = "/home/wojciech/Desktop/data/net" + str(i) + ".paj"
        p = PajekReader(s)
        graph = p.getGraphData()
        # Now we have the GraphData object loaded.
        # Count the number of alters.
        numAlt = g.getAlterCount();
        # Write this number to file.
        w.write(str(numAlt) + "\n")
# End the program.
```

Customizing the File Menu

Since Egotistics is implemented through the use of *Jython*, the file menu system has been implemented through the use of text files and is fully customizable. Indeed, with simple scripting similar to that used in the console and the editing of two text files, the entire file menu may be changed.

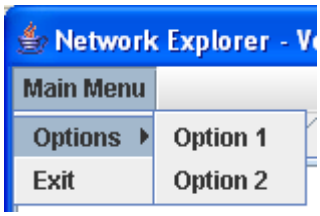
Rather than being coded into the software, the file menu is loaded at during the Egotistics startup from two separate files, both located in the program directory. The first file, *menu_labels.txt*, contains the various labels used within the actual software. The menu structure is organized using four different notations, all of which are either one or two characters long and precede the actual file label. They are:

`+`	Indicates a new file menu within the actual window, similar to File, Edit, View, and other similar menus in popular Windows applications.
`++`	Indicates a sub menu. This is like a secondary drop-down menu within held within a main menu (+).
`*`	A clickable menu button. This is a button that runs a command when clicked, and is stored as an button in one of the main menus.
`**`	Like above, but this is a button stored in a submenu.

Using the above specifications, it should be clear that a submenu (`++`) must always be held within a menu (`+`), and that submenu buttons (`**`) should only follow a submenu declaration (`++`). For example, the code below,

```
+ Main Menu
++ Options
** Option 1
** Option 2
* Exit
```

will yield the following type of menu:



To add further customization to the menus, one can label a button as `-- SEP --` to create a separating line between buttons in a menu or submenu.

A second file, *menu_commands.txt*, contains the code to be executed whenever a button in a menu is clicked. Note that menu titles, submenu titles, and separators cannot run code. Code is written like code within a console. The only difference is that the code does not support multiple lines: rather than creating new ones, each line should end with a semi-colon. Each line corresponds to the object represented in the corresponding line of the *menu_labels.txt* file.

For example, if we want "Option 2" to count the number of components in each network of *CP*, the third line of the *menu_labels.txt* file should be: `Components.countAll(CP).`

Similarly, if "Exit" should say "Goodbye..." and then exit the program, the fifth line of *menu_labels.txt* should be: `print "Goodbye..."; UI.dispose().`

Customizing the Toolbar

One can also customize the toolbar in a similar way as modifying the file menus. Two files are located in the program directory, *toolbar_commands.txt* and *toolbar_icons.txt*. This systems works in the same manner as the file menu one: the n^{th} line of the first file corresponds to the code executed when the nth toolbar button from the left is clicked.

The list of toolbars is included in the second file, *toolbar_icons.txt*. Each line of this file contains two pieces of information, the toolbar button's title followed by the icon's image location. Icons are located in the img folder in the program directory, and each icon is a 16-pixel-wide square, saved in GIF format. Egotistics will resize the toolbar to whatever size the button icon is, so it is recommended that all icons are of this size. The two pieces of information are split by a semi-colon.

While not used in the toolbar included in the software, the *img* directory contains a *generic.gif* icon for scripts.

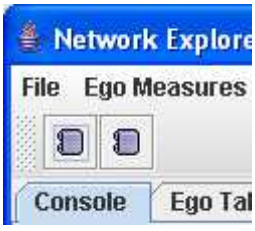
For example, if one was interested in making a toolbar with two buttons, one that prints “Hello” in the console and the other leaves the program, the *toolbar_commands.txt* file would contain the following information:

```
print "hello"
UI.dispose()
```

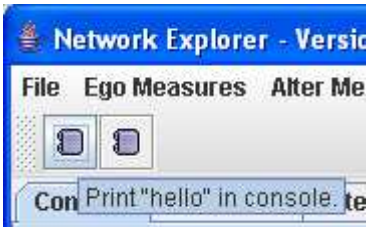
The *toolbar_icons.txt* file would have the following:

```
Print "hello" in console.; /img/Generic.gif
Exit the program.; /img/Generic.gif
```

The text included appears only when one moves the cursor over the button in the toolbar. Below is a screenshot of the new toolbar.



Standard view.

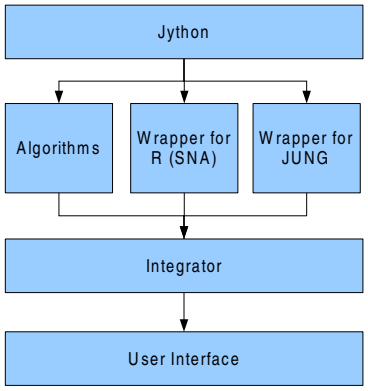


With tool tip.

Software Architecture

Egotistics's goal is to aggregate information from other programs and implements algorithms currently not available in any *API*s. The software is also built to act like a batch processor: it will apply the algorithms and make measurements of network statistics and structural information on all networks within a project automatically.

The software itself is built using *Java* and implements *Jython* as the standard interface. While a graphical user interface is available, the buttons and menus ultimately call *Jython* commands, which then interact with the project currently loaded into the network. The entire program architecture is shown in the diagram below:



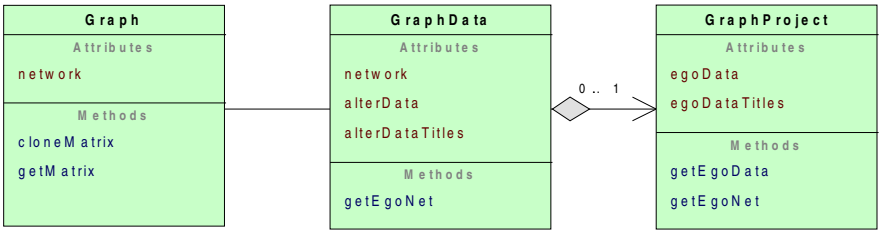
In this case, *Jython* is the main interface between the project and network analysis tools within the program. Each measurement tool is contained within a *Java* class and implemented as a static method. This allows one to call the method in *Jython* without having to instantiate any new classes. The current Egotistics version will include custom algorithms and wrappers for specific methods in *R* and *JUNG*. Egotistics is also open source, and has been designed to incorporate new classes and algorithms with ease. This is described in a later section.

To facilitate incorporation of new measurements and statistics into a project, an *Integrator* class was created, which contains methods for adding ego-level and alter-level statistics. These are described in full detail in the project's *JavaDocs*, and an example of how to create ego- and alter-level code is given later in this manual.

The data within a project is organized in a *GraphProject* object. A *GraphProject* object contains ego-level statistics and attributes, which includes network-level information like number of alters or network density. The object also contains a collection of *GraphData* objects, which act as a repository for alter-level characteristics of each network, as well as the network itself.

While the software treats each network as an adjacency matrix, for convenience, every network is stored within a *Graph* object, which allows it to be loaded easily into the software, and allows for a few simple capabilities, such as cloning the matrix.

The UML schema outlining how a project is organized is shown below:



Project Files

When you save a project to a file, a number of files are created (or overwritten!). The most important file is the *NXE* file, which stores all of the project information. The *NXE* file is a standard text file with file locations listed at each line. The first line represents the location of the ego *CSV* file, the second is the alter *CSV* file, and the third line onward contains the locations of the network files themselves.

Creating Your Own Classes

If Egotistics does not contain the algorithms required for your own analyses, extending the software is not difficult. A typical class for providing measurements contains private methods to help with the measurements, as well as public methods that calculate the measurements at the ego level, alter level, or both.

Supposing you would like to implement a “Some Algorithm”, you could create a class with that name. Two required imports are the *ArrayList*, which is required for alter-level statistics, and the Egotistics *Integrator*, which connects the class's output to the specific *GraphProject*.

```
import edu.netlab.util.Integrator;
import java.util.ArrayList;
public class SomeAlgorithm {
```

The name of the method within the class does not matter, though it is recommended that implementations of algorithms use static methods. Doing so allows one to call the class using *Jython* without instantiating it first. The method below accepts only one argument, the *GraphProject*, though more parameters are allowed. When looking for K-Plexes of size N or greater, for example, such a method would be called *getEgoMeasures(GraphProject gp, int n, int k)*.

```
public static getEgoMeasures(GraphProject gp) {
```

In the case of egos, there should be one measurement per ego, and each is stored as a String. This is meant to facilitate the reading and writing of data.

```
String[] output = new String[gp.getNetCount()];
```

After the above is instantiated, one should have a loop or some code that will assign the required values for each ego. Egotistics is open source and fully commented, so taking a look through the package *edu.netlab.algorithms* can help clarify what goes in this section of code.

One the calculations are done, however, the Integrator can be used to assign the values to the proper location of the *GraphProject* object. The *addEgoData()* method only has two parameters, a *String* representing the title of the data in the ego table, and the actual output of the measurements.

```
Integrator.addEgoData("Header", output);
}
```

A similar process is behind developing alter-based measures. However, rather than creating an array of measurements, an *ArrayList* is used instead. In this case, each member of the *ArrayList* is an one-dimensional *String* array, with measurements done for each alter in the ego network.

```
public static getAlterMeasures(GraphProject gp) {
ArrayList allMeasures = new ArrayList();
```

To get measurements in, one much loop through each ego, and get the proper *GraphData* object representing that ego's network and alters.

```
for (int i = 0; i < gp.getNetCount(); i++) {
GraphData gd = gp.getData(i);
```

Once gd, the object representing the current ego network, has been obtained, one can create an String array that will contain one empty data slot for each alter in the network.

```
String[] measures = new String[gd.getAlterCount()];
```

Next, a second loop should be created where each alter is assigned the measure required.

```
for (int j = 0; j < gd.getAlterCount(); j++) {
// Your code.
}
```

Again, the code inside the loop depends on the specific algorithm. See the Egotistics source code for examples. Once the entire String array is populated, the array is added to the *ArrayList*.

```
allMeasures.add(measures);
}
```

Finally, once all the measures are in, the data title and data is sent to the *Integrator* class, which adds the new data to the current *GraphProject*.

```
Integrator.addAlterData("Header", allMeasures);
}
```

Printing to the Console and Status Bar

Aside from appending information a *GraphProject* object, it is also possible to print information in the console and status bar within Egotistics. In both cases, one actually sends information to the `System.out` stream. While this usually prints to a terminal or command line window, Egotistics ensures that anything

sent to its `System.out` stream will be printed in the console. One must note, however, at the console itself uses HTML formatting. As such, information sent to the console could be represented as plain text, or may use HTML tags.

If one is interested in printing on the status bar, one must precede the text sent to the `System.out` stream by `(!)`. Thus, the following Java code will be printed on the status bar:

```
System.out.println("(!) This will update the status bar.");
```

The status bar does not support HTML formatting. The console and status bar both disregard new line characters (the console requires a `<br>` or `<p>` tag, while the status bar only supports one line of text), so one can use either the `print()` or `println()` functions when passing information to be printed.

Advanced Python Scripting

Egotistics allows for advanced Python scripting using functions, classes, and other tools found in both Java and Python. While Java-based coding was covered above, the Integrator class used is also available within Python. Combining this with GUI-based Java functions allows one to easily create new methods and implement functions not currently available within Egotistics without having to recompile it using Java.

For those interested in learning more, it is recommended they visit the Jython website (http://www.jython.org) and read the JavaDocs available for this software.

A Basic Example

At the ego-level, a *GraphProject* object has a simple function called *getEgoData(<ego>, <column>)* that returns a String object with the required data. The column order corresponds to the other shown in the actual *Ego* tab in the software.

To obtain information on a specific ego (e.g. the network or alter-level data), one needs to get a *GraphData* object from the *GraphProject*. This is done through the *getGraphData(<ego>)* method, which returns the required *GraphData* object. This object then has a *getAlterData(<alter>, <column>)* method that returns information about alters.

Using the above, it is possible to make calculations and obtain information through Python. Inserting it back into the *GraphProject* object requires the use if the *Integrator* class. This is a static class that handles any data insertions into projects, and has the two important functions:

`addAtEgoLevel`	Contains three parameters in the following order: *GraphProject object*, *Header/Title of column*, *String array representing data*. The order of the array corresponds to the order of the egos in the graphical user interface.
`addAtAlterLevelAll`	With the following parameters: *GraphProject object*, *Header/Title of column*, *ArrayList object containing String arrays*. In this case, each

ArrayList element contains a *String* array which has data corresponding to each alter.
Note that a similar function exists which accepts a String[][] array rather than an ArrayList.

A simple example at the ego level is presented below, where a new column is added to the data with an integer for each ego. You may run the code below using the *Scripting Window*.

```
egolist =[]
for i in range(0, CP.getNetCount()):
        egolist.append(str(i))
Integrator.addAtEgoLevel(CP, "New", egolist)
```

Advanced Example: Egos

Another example is presented below. In this case, the density is calculated for each ego using the standard *Egotistics* functions, but a second column is added. This second column has the value "A" present if the density is below 0.5, while "B" if the density is higher.

```
# Calculates density.
NetStats.density(CP)
# The value below is the index of the density column.
# It is the newest column added.
densIndex = CP.getEgoDataTitlesLength() - 1
# Now a new list is created.
egolist =[]
# Loops through each ego.
for i in range(0, CP.getNetCount()):
        val = "B"
        if (float(CP.getEgoData(i, densIndex)) < 0.5):
                val = "A"
        egolist.append(val)
# Now it adds the information using the Integrator class.
Integrator.addAtEgoLevel(CP, "Density Categories", egolist)
# Needed to update user interface.
UI.showProject(CP)
```

Advanced Example: Alters

To add alter-level functions, it is easiest to use a String[][] object. The first array contains a set of

String[] objects, each one corresponding to the alters in an ego network. The example below simply appends a column full of periods to the data set.

```
egolist = []
for i in range(0, CP.getNetCount()):
          alterList = []
          gd = CP.getGraphData(i)
          for i in range(0, gd.getAlterCount()):
                    alterList.append(".")
          egolist.append(alterList)
Integrator.addAtAlterLevelAll(CP, "New", egolist)
UI.showProject(CP)
```