

Zoomgraph version 0.2 (alpha)

Eytan Adar & Joshua Tyler

1. Zoomgraph features

This tool is:

- A zoomable interface to large graphs. Zoomable means you can smoothly zoom in and out and easily move between nodes. It's not exactly what you would expect and is a little hard to explain, you'll have to try it.
- A database driven system. Nodes and edges have features that you can query and use to control what gets displayed (e.g. show all the nodes and interactions for yeast genes that have a metabolic function, show all the email communications between two departments).
- Facility to describe and manage subgraphs
- Robust language for selecting and managing nodes and edges
- Writes out SVG and jpeg
- A few layout algorithms
- Talks to R (some functions, but it's easy to add more)

This tool isn't:

- A replacement to UCINET, Pajek, whatever else you may be using. It has some overlapping features, but really it's meant to be an additional resource.
- Anywhere near perfect. If we had to do it again there are lots of things we would have done differently (written a parser, thought the language through, better object model, etc.)
- Complete. There are certainly features we would want to add or extend to make this truly useful (labeling on saved files, complete R integration, etc.).

2. Getting Started

Installation

You're going to need 3 things:

- The Java runtime (version 1.4). You can get the SDK, but all you really need is the JRE. <http://java.sun.com>
- The Zoomable class file and sample files. You can download the zip file at: <http://www.hpl.hp.com/shl/projects/graphs/> and unzip it somewhere.
- A bunch of 3rd party class files. I've packaged them all up for you at the URL above. Please read the various copyright notices. Specifically, you're getting the Colt libraries (used for some of

the visualization), HSQLDB an embedded database, Rserve (modified the included jar so that the classes sit in a package), the Batik libraries for SVG output (put all the separate jar files into one), and finally the Jazz visualization library (also modified this slightly because of some thread safety issues. The modified copies just catch exceptions a little better). Put those jar files in the directory where you put the zoomable jar.

Zoomgraph is launched from the command line, so start up a command prompt. You're going to want to set the CLASSPATH so that it includes all the JAR files. We've included a file called setvarbs.bat which you can run to set everything for you.

Running

To build a database you can run:

```
java com.hp.hpl.zoomgraph.DBServer db_name db_definition_file
```

To run the browser do:

```
java com.hp.hpl.zoomgraph.ZoomableGraph db db_name
```

There are a few other ways to run zoomgraph. You can type:

```
java com.hp.hpl.zoomgraph.ZoomableGraph db edge_def_file
```

This will take the edge definition file: *edge_def_file* (see Section 5 for formatting instructions) and will generate a temporary database (overwriting the previous temporary database). You can also work without a database by doing:

```
java com.hp.hpl.zoomgraph.ZoomableGraph edge_def_file
```

Finally, you can also run zoomgraph as an applet (see Section 5).

Tutorial

Let's start with a simple example. There is a sample database (sample.database) in the zip file. It includes about 400 nodes and 700 edges. Take a look at it to get a sense of what goes into a data definition file. But don't get intimidated, almost none of it is required.

After setting your classpath (see above), type the following to transform the text file to an actual SQL server:

```
java com.hp.hpl.zoomgraph.DBServer sample sample.database
```

then do:

```
java com.hp.hpl.zoomgraph.ZoomableGraph db sample
```

You'll see something that looks like Figure 1.

The graph that popped up represents a corporate communication network. Each node represents an employee (with a department property), and each edge represents communication between two employees (with a frequency property on the edge indicating the number of undirected communications).

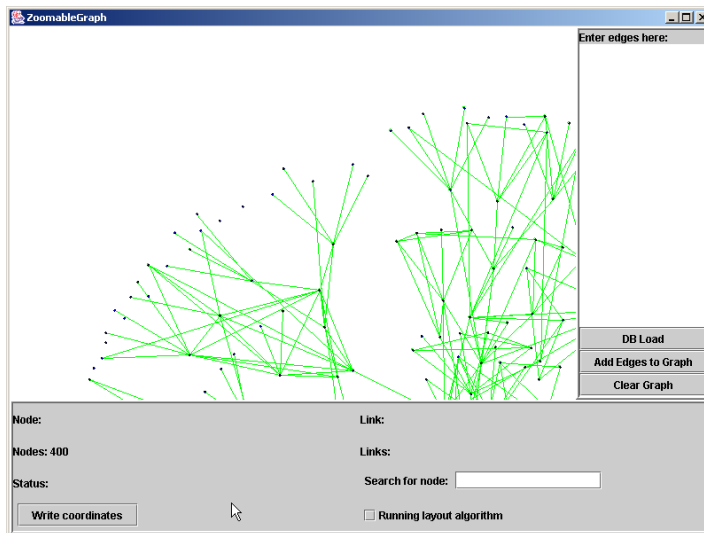


Figure 1

Try moving around in this space. If you hover over a node or edge you can see some details pop up. If you click on the node it will center in the display. Clicking on an edge will bring both end points into view dynamically. Left clicking and dragging on the background will allow you to move the display around. Right clicking and moving the mouse will zoom you in and out of the display.

Type "node1" into the search box and hit enter. The display will automatically shift to center on that node.

Ok, now back in the command prompt where you started zoomgraph you should see a prompt that looks like this ">." Here you can type whatever commands you want to manipulate the graph. Type "center"

and hit enter. The display will automatically center to include all the nodes. Your display should look like *Figure 2*.

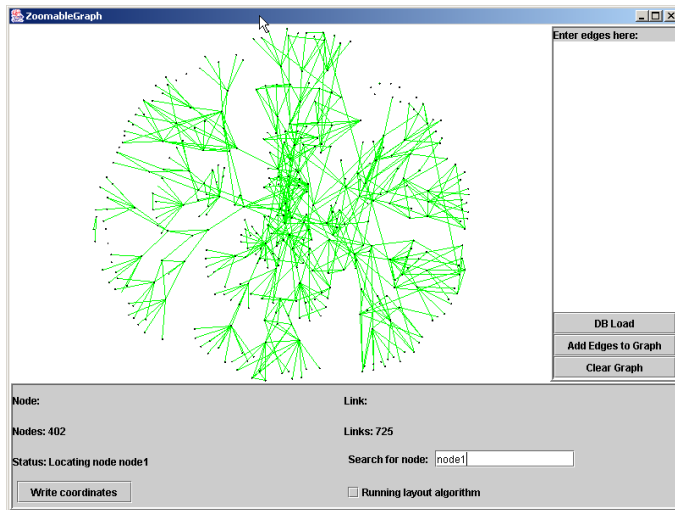


Figure 2

Let's make the nodes a little bigger so we can see them a little better. Type `nodesize 10` to make all nodes 10 pixels. If things don't immediately change on the display for any of these commands just type `redraw.`

Nodes can either be selected by name or through a SQL query. For example, try typing: `nodecolor red node5,node6` This will make nodes 5 and 6 red. Our sample database has other properties on nodes. Specifically, nodes here have a department. To select nodes by a SQL query you can just type what you would after the WHERE clause. For example, `nodecolor black dept = 'dept5'` will set all the people in department 5 to a black color. Some commands just assume you want all the nodes if you don't enter a list, otherwise the character `*` means all nodes (or edges if it's an edge command).

Edges are accessed in a slightly different way. Edges have names that are the start and end nodes. For example, `edgecolor red node67-node76` changes the edge between person 67 and 76 to red. You can also access edges by SQL queries. As mentioned earlier, edge in this case have an attribute called freq (frequency). So if we wanted to hide edges where the communication frequency was under 100 we would type: `hideedges freq < 100` The `->` also implies directionality. If the database indicated directions (which this one doesn't) you could talk about `node67->node76`, `node67<-node76`, or `node67<->node76`.

The last mechanism for accessing edges is by defining node sets. Let's say we only care about communications between dept 4 and 9. Let's hide everything: *hideall*. Then show only the nodes in departments 4 or 9: *shownodes dept = 'dept4' OR dept = 'dept9'*. Finally, we can change the color for inter-departmental edges by typing: "*edgecolor red {dept = 'dept9'}-{dept like 'dept4'}*" This command tells the Zoomgraph to find all nodes in dept 4 and all nodes in department 9, and through some SQL magic that goes on in the background it will find all edges between them (in this case only one). We can also do "*edgecolor blue {dept = 'dept9'}-{dept = 'dept9'}*" to just get intra-departmental links blue. You should see something like *Figure 3*.

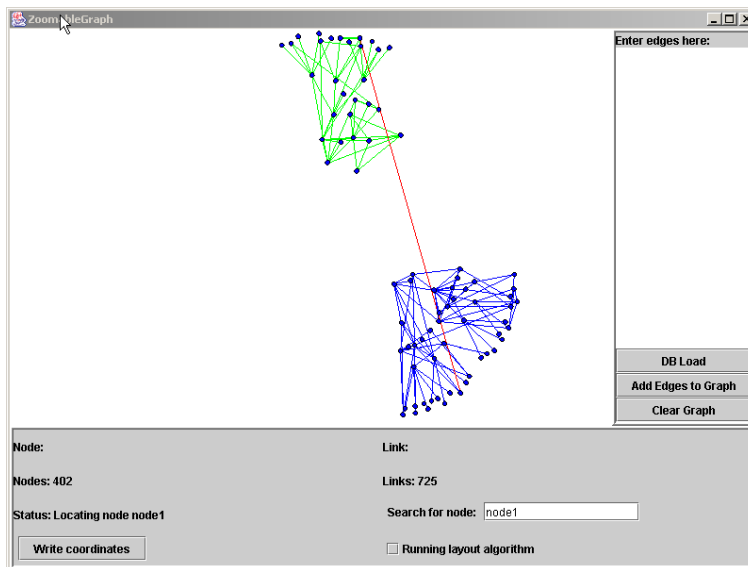


Figure 3

The Zoomable system also contains a number of analysis modules to simply basic tasks (calculating graph metrics, etc.) These are described in much more detail elsewhere, but just to give you a flavor try this... First, reset the graph to it's starting state. Type: *showall*. Then type: *edgecolor green* and finally *nodecolor blue* (you should see the same thing as before). We do this because when nodes and edges invisible it is the same as if they weren't there and are not counted in various calculations. Type "*analysis density **" This should calculate the density of the graph (.00827...).

Other analysis modules do more interesting things. For example, *colorize* will color nodes and edges by different features. Try typing "*analysis colorize nodes dept.*" Each node will now be a different (random) color. The *colorize* function will also generate a bunch of subgraphs. Then try "*analysis colorize edges freq linear*" which will

assign a color over a linear range (from blue to red) based on the frequency of communications.

Subgraphs let us bundle a related edges and nodes together and give them a name. You can type `"sg list"` to see a list of named subgraphs. Running the `colorize` command gave us a different subgraph for each department and one main one `"AUTO_nodes_dept_all"` that holds all the subgraphs (yes, subgraphs can be nested). Operations that work on nodes and edges will also work on named subgraphs in the same way. You can type `"hidenodes AUTO_nodes_dept_dept1"` to hide all the nodes in that subgraph. `"sg details subgraphname"` will tell you which nodes and edges and nested subgraphs are part of the named subgraph. Nested subgraphs are referenceable by using a `.*` at the end of the subgraph name. For example `"hideall AUTO_nodes_dept_all.*"` will hide each individual dept subgraph contained in `AUTO_nodes_dept_all` (this command isn't all that interesting since you could do without the `.*`, but try `"sg details AUTO_nodes_dept_all.*"` and `"sg details AUTO_nodes_dept_all"` to see the difference).

Our last example that combines everything and adds some new twist. Let's visualize all the pairwise connections between departments, one at a time. We can do this by using the `foreach` loop. The `foreach` loop takes two arguments a variable and a set. For each element in the set, the variable will be set to that element and the loop will be done. So try this:

```
> foreach sg1 AUTO_nodes_dept_all.*
  > foreach sg2 AUTO_nodes_dept_all.*
    > hideall
    > shownodes sg1
    > shownodes sg2
    > hideedges *
    > showedges {sg1}-{sg2}
    > redraw
    > pause 2000
    > .
  > .
```

What we're doing is looping over all the colored subsets. The variables `"sg1"` and `"sg2"` are set by the `foreach` loop. The system knows that you're talking about subgraphs and not nodes or edges because of the naming convention. If you wanted all the edges in the

subgraph you would do "foreach edge1 AUTO_nodes_dept_all"
similarly all nodes are "foreach node1 ..."

Hopefully this shows you enough to get you excited about playing with the system. Below are the commands that you can use right now.

Listing Objects

Most of the commands you'll see below allow you to enter a list of objects that you are applying the command to. The tutorial gave you some examples of how to call graphs and edges, but a quick summary may be helpful.

Nodes can be referred to by their name.

- You can submit a list of comma delimited nodes when you have the option to list. For example "node5,node6,node7" is a valid list.
- You may also refer to nodes by means of a SQL like query acting on node properties. For example you may pass "size < 50" as the list to something acting on nodes (this will select nodes that are under 50 pixels in size). SQL queries can be complex. For example: "size < 50 OR (xloc > 500 AND yloc < 400)"
- You cannot mix SQL style selects and standard node names in a list. If you want to be clever you could add the nodes to a subgraph, generate a node view (see commands), and then apply the SQL query to the view.
- * refers to all nodes
- If you have a subgraph defined with nodes in it, any node command applied to that subgraph will act on the nodes held within the subgraph.

Edges are more complex.

- If you had an edge between node5 and node6 you could refer to it as node5-node6 or node6-node5. If you have multiple edges between node5 and node6, edges also have a numerical id property that you can use as the name. You can list edges the same way as nodes: "node5-node6,node8-node9"
- If edges are directed you replace the "-" character by "->" "<->" or "<-" For example, "node5->node6,node8<->node9" is a valid way to refer to the directed link between node 5 and 6, and any link between node8 and node9.
- Finally, you can reference edges through a set-to-set operation. A node set is defined within the curly braces..."{}" Inside you can place any node list as described above. This can either be a

list of nodes or a SQL selection (or subgraphs). For example, you could say "{xloc<500 AND yloc < 500}-{xloc > 500 AND YLOC > 500}" to get all the edges going from one nodes in one quadrant to nodes in the other.

- * refers to all nodes
- As before, if a subgraph contains edges, any edge command applied to that subgraph will act on edges within the subgraph.

3. Getting Your Data in: DBServer

The DBServer program will initialize your database for you. It takes as input a database name and a database description file (see above on the command line). The description file has two components a list of nodes and a list of edges.

The node definition section starts with the line: "nodedef> name ..."

The only required column for nodes is a name, which needs to be a string. So a valid node definition line would be: "nodedef> name VARCHAR(256)" After that you can simply put down a list of nodes, one per line that match the specification in the nodedef line (in this case all you need is a name). Similarly, edges are defined through an "edgedef>" line. An edgedef line must define two columns n1 and n2 (the start and end nodes of the edge). A valid database description file is then:

```
nodedef> name VARCHAR(256)
A
B
C
edgedef> n1 VARCHAR(256),n2 VARCHAR(256)
A,B
B,C
A,C
```

Which basically represents an undirected graph with three nodes and three edges.

In addition to the required columns there are a number of optional ones for both nodes and edges. These are created for you by the DBServer if you don't do it yourself. Note that the def describes what comes next. You have to have the same number of columns in each of your node and edge lines as you did in your definition lines.

You may choose different defaults, but try to use the same types – DOUBLE, INT, etc. that are described here. The Zoomgraph makes certain assumptions about what's in a database. Node definition lines may include:

- XLOC DOUBLE DEFAULT 500 – which is the x location of the node
- YLOC DOUBLE DEFAULT 500 – the y location of the node
- VIS BIT DEFAULT TRUE – is the node visible (true = yes)
- COLOR VARCHAR(32) – the color of the node... be careful about using r,g,b values since the comma delimiter may break the database load
- FIXED BIT DEFAULT FALSE – is the node fixed in place (true = yes)
- SHAPE TINYINT DEFAULT 0 – the shape of the node, currently 0 = circle, 1 = square. May add more later. You won't be able to change this dynamically right due to constraints of the visualization.
- SIZE DOUBLE DEFAULT 2 – the size of the node

Edges can have the following properties:

- VIS BIT DEFAULT TRUE – is the edge visible (true = yes)
- COLOR VARCHAR(32) – the color of the node... be careful about using r,g,b values since the comma delimiter may break the database load
- WIDTH DOUBLE DEFAULT .3 – the width of the edge
- WEIGHT DOUBLE DEFAULT 1 – the weight of the edge
- DIRECTED TINYINT DEFAULT 0 – the direction of the edge. This doesn't work all that well yet. 0 means undirected, 2 means "n1->n2", 3 means "n1<-n2", and 4 means "n1<->n2"

Beyond these basics everything is fair game. You can add whatever columns and properties you want and then use them to control your visualization. For example, let's extend our basic definition above to indicate node size and a new column called city, and edges will have a number representing the number of planes (totally fake):

```
nodedef> name VARCHAR(256), SIZE DOUBLE DEFAULT 2, CITY
VARCHAR(256)
A,10,new york
B,6,boston
C,4,san jose
edgedef> n1 VARCHAR(256),n2 VARCHAR(256),ROUTES INT DEFAULT 0
A,B,40
B,C,30
```

4. Commands

Display Commands

center [list]

Centers the display to include all nodes (if no argument is given), or only those that are in the list. This will only center on visible nodes.

Note: You may notice an exception sometimes when you do this even though the display does the right thing. There seems to be some race condition in Jazz.

centerall [list]

Centers the display to include nodes (all or in list), visible or not.

freeze

Freezes the display. Changes will not appear on the screen until you unfreeze it.

unfreeze

Unfreezes the display

rq [low|medium|high]

Sets the render quality of the visualization to one of three states. If you just type "rq" it will tell you what the current state is. The quality only applies to a display that has stopped changing. Moving around may cause the display to shift to a different rendering quality (see *rqi*). Default: low

rqi [low|medium|high]

Same as *rq*, but sets the interactive state. This sets the rendering quality when the display is changing. Default is also low. Changing this may degrade performance in display related features since rendering will take longer.

background color

Sets the background color to color (see section 5 for more information about colors).

redraw

Re-renders the display. Sometimes this is necessary to get the display to sync up with certain commands.

iw+

opens the information window. As you mouse over nodes and edges the information window reflects details about the objects. The information is the same as the node/edge details command.

iw-

hides the information window

General Graph Commands***hideall [list]***

Hides all graph objects if no argument is specified. Otherwise only hides the objects in the list. This can be a mix of edges and nodes.

hideall [list]

Shows all graph objects if no argument is specified. Otherwise only shows the objects in the list. This can be a mix of edges and nodes.

muteall [list]

Mutes all graph objects if no argument is specified. Muted nodes are shown in the muted color (default gray). If a list is supplied only mutes the objects in the list. This can be a mix of edges and nodes.

unmuteall [list]

Unmutes all graph objects if no argument is specified. If a list is supplied only unmutes the objects in the list. This can be a mix of edges and nodes.

mutecolor color

Sets the muted color for muted nodes/edges

directed

Sets the graph mode to directed. If your edges are undirected you won't see anything different. If there is a direction, you should see some arrows (this only sort of works)

undirected

Sets the graph mode to undirected.

commitoff

Turns off database commits. Changes made to nodes/edges will *not* be committed to the database. Default: on

commiton

Turns on database commits. Changes made to nodes/edges will be committed to the database. Default: on

Output/Scripting Commands***savejpg file_name***

Outputs the current visualization to the specified jpeg file

savesvg file_name

Outputs the current visualization in svg format to the specified file

savelog log_name

All commands that are typed will be saved to the log specified by log_name. Using this you can save what you do and re-run or modify it later to make macros.

loadlog log_name

All commands in the log file (log_name) will be executed against the current environment. This can be used to reply macros that you have saved earlier.

stoplog

Stops logging commands

savecsv file_name SQL_query

Will generate a file (specified by file name) in CSV format for the database columns you are interested in

Example: `"savecsv foo.csv select name,degree from nodes"`

Will save the name and degree columns to the file foo.csv

pause milliseconds

Pauses the system for some number of milliseconds. This is useful when running a script and you don't want the display to refresh too quickly.

foreach variable list

The foreach loop lets you loop over a list of objects. The variable naming convention is to start the name with "node" if you want the nodes, "edge" if you want the edges, and "sg" if you want the subgraphs. So you could do: "foreach node1 somesubgraph" to talk about the nodes in the subgraph or "foreach edgefoo somesubgraph" to references the edges. Any instance of the variable name gets replaced by an element in the list each time the foreach loops. To close a foreach statement just type a period (".") on it's own line.

Node Commands

node details [list]

Gives you some details about the node (same as you would see in the information window).

hidenode(s) [list]

Hides all nodes if no argument is specified. Otherwise only hides the nodes in the list.

shownode(s) [list]

Shows all nodes if no argument is specified. Otherwise only shows the nodes in the list.

mutenode(s) [list]

Mutes all nodes if no argument is specified. Otherwise only mutes the nodes in the list.

unmutenode(s) [list]

Unmutes all nodes if no argument is specified. Otherwise only unmutes the nodes in the list.

fixnode(s) [list]

Fixes all nodes if no argument is specified. Otherwise only fixes the nodes in the list. Fixed nodes will *not* be moved by the layout algorithm.

unfixnode(s) [list]

Unfixes all nodes if no argument is specified. Otherwise only unfixes the nodes in the list.

nodecolor color [list]

Sets the node color of all nodes (if no argument is given) or just those listed.

hidedis [list]

Hides nodes (all if no argument, or from the subset of list) that do not have any visible edge going to them.

nodesize size [list]

Sets the node size of nodes (or all if no argument) to size

Edge Commands***edge details [list]***

Gives you some details about the edge (same as you would see in the information window).

edgecolor color [list]

Sets the edge color of edges (or all if no argument) to color. Sometimes this requires an explicit redraw to be called.

edgewidth width [list]

Sets the edge width of edges (or all if no argument) to width. Sometimes this requires an explicit redraw to be called.

hideedge(s) [list]

Hides all edges if no argument is specified. Otherwise only hides the edges in the list.

showedges(s) [list]

Shows all edges if no argument is specified. Otherwise only shows the edges in the list.

muteedges(s) [list]

Mutes all edges if no argument is specified. Otherwise only mutes the edges in the list.

unmuteedge(s) [list]

Unmutes all edges if no argument is specified. Otherwise only unmutes the edges in the list.

unkink

Some layout routines cause bends in the edges, you'll want to run this to restore the straight line mode.

Subgraph Commands

Subgraphs are hierarchically structured objects that contain nodes, edges, and references to other subgraphs. Figure 4 is a representation of three subgraphs, sgA, sgB, and sgC. The subgraph sgA directly contains nodes A and B, and edges A-B. It also contains a pointer to subgraphs sgB and sgC. All commands and lookups applied to sgA will be recursively applied to these embedded subgraphs. For example, setting the node color to red for sgA will cause not only nodes A and B to become red but also nodes D and E. This is usually the expected thing, but be careful when applying operations like delete as they will be applied to lower level subgraphs as well. For example deleting node D from sgA will actually cause its removal from sgB. The commands for subgraph manipulations are as follows:

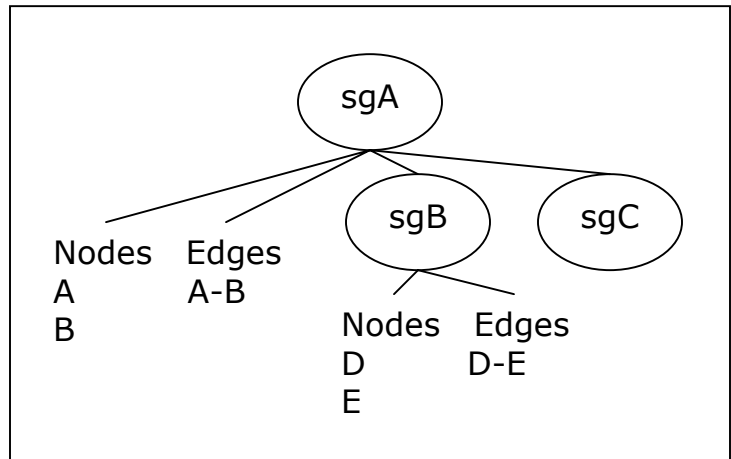


Figure 4

sg subgraph_command [optional arguments]

Performs a subgraph command

- ***list*** – lists all the named subgraphs
- ***details name*** – gives information about the subgraph named name
- ***save filename*** – saves the subgraph definition to filename. Section 5 describes the file format for the subgraph files.
- ***load filename*** – loads the subgraph definition from filename
- ***create newname*** – creates a subgraph with the name “newname”
- ***delete [list]*** – deletes the named subgraphs. If no argument is given it will happily delete all the subgraphs from memory (so be careful).
- ***nview name*** – creates a database view including only the nodes in the subgraph named “name.” This is useful if you want to do some heavy duty SQL manipulation.
- ***evview name*** – creates a database view including only the edges in the subgraph named “name” Useful for SQL manipulation.
- ***dropnview name*** – drops the node view for the given subgraph. You should have called nview before calling this.

- **dropeview name** – drops the edge view for the given subgraph. You should have called eview before calling this.
- **addnode(s) name list** – adds the listed nodes to the named subgraph. If the node is already contained in either the top level subgraph or any of it's contained subgraphs the node will not be added.
- **addedge(s) name list** – adds the listed edges to the named subgraph. If the edge is already contained or is nested it will not be added. Adding an edge also causes the end points to be added.
- **addsg(s) name list** – adds the listed subgraphs to the named subgraph. Use caution in applying this command. The system does not check for cycles in subgraphs. So if subgraph A contains B, and B contains A, certain operations will freeze the system.
- **rmnode(s) name list** – remove the listed nodes to the named subgraph. It will do a recursive removal, so if the node is contained in a contained subgraph it will be removed from that lower level subgraph.
- **rmedge(s) name list** – removes the listed edges to the named subgraph. Same rules as rmnodes.
- **rmedge(s)+ name list** – removes the listed edges to the named subgraph and removes any nodes that are now disconnected.
- **rmsg(s) name list** – removes the listed subgraphs to the named subgraph.
- **copy name1 name2** – copies the contents of subgraph name1 to subgraph name2.

Analysis Commands

analysis analysis_command [optional arguments]

Performs an analysis_command. Valid commands include:

- **concom**– Figures out all the connected components of the network. It will generate a subgraph called cc_all. Each connected component becomes a subgraph of cc_all. It will be named cc_node_size, where node is the name of some node in that connected component and size is the size of the connected component.
- **colorize table column [random|linear]**- The colorize command will generate a coloring on either the nodes or edges

table depending on some property (the column). The number of colors it will pick depend on the range of the column. For example, if column is a boolean column (true/false), there will be two classes and therefore two colors. Random coloring will assign a random color to each class. Linear (blue to red) usually only makes sense when the column is a number. For example if you had an edge weight column you could do "colorize edges weight linear" and have low weight edges be blue, and high weight edges be red. This algorithm will also generate a set of subgraphs. The subgraph AUTO_table_column_all will hold all the subgraphs generated where each subgraph represents a different color coded class.

- **clustCoef [list]** – calculates the clustering coefficient on the list of nodes
- **density [list]** – calculates the density on the list of nodes
- **diameter [list]** – calculates the diameter on the list of nodes
- **avgPathLength [list]** – calculates the average path length on the list of nodes (really slow for long lists)
- **symmetry [list]** – calculates the symmetry on the list of nodes
- **rservehost hostname** – sets the name of the machine hosting R for doing R based analysis. You will need to install Rserve and have a machine running R somewhere (follow the instructions at: <http://stats.math.uni-augsburg.de/Rserve/>)
- **betweenness [list]** – using R, it will find the betweenness of a set of nodes and place the results in a betweenness column in the nodes table. This requires a running R with Rserve
- **degree [list]** – using R, it will find the degree of a set of nodes and place the results in a degree column in the nodes table. This requires a running R with Rserve. Yes, we could calculate this some other way, but it makes an easy test for making sure the Zoomgraph<->R stuff works.

Layout Commands

layout layout_command [extra]

These are various layout commands.

- **fruch** – Fruchmen-Reingold layout
- **random** – random layout
- **circular** – regular circular layout
- **kamada** – Kamada-Kawai layout
- **hierarchy column numlayers** – still in alpha. It will layer nodes depending on the value of column into num layers. You'll want to run unkinck when you're done with this

Power User Commands

Backdoor

Typing "backdoor" puts you in direct contact with the SQL server. Your prompt will change to "b>" and any commands typed after that will be directly routed as SQL. You should be careful if you choose to use this. The results of select commands will be dumped to the screen. Typing **backdoor** again when in this mode will put you back in the regular mode. Typing **quit** will exit zoomgraph not just the backdoor mode.

select/SELECT sql_stuff

Does a SQL select and dumps the results.

CREATE VIEW sql_stuff

If you're into doing some heavy-duty table merges and selects you can use this. It will route this type of command to the SQL server. You should remember to drop this if you don't want it to be persistent.

DROP VIEW

Lets you drop the view you created. Routed directly to the SQL server.

temp sql_stuff

This will run a full query on the database and place the resulting nodes and/or edges into the subgraph __temp. This comes in handy if you're doing weird selects and joins. You can then copy the nodes/edges out of the temporary subgraph. For example: the command "temp SELECT * from nodes where xloc < 500" will load up the __temp with nodes that are on the left side of the display. The routine basically sees if the table you're querying on has a column called "name" which indicates that you want nodes, and/or columns named n1 and n2. You can get both through different select and view operations.

5. Additional Information

Colors

Colors are defined by a comma delimited numbers representing red, green, and blue. There can be no space between them. For example the color red is "255,0,0." There are also a number of predefined named colors: black, blue, cyan, darkGray, gray, green, lightGray, magenta, orange, pink, red, white, and yellow.

Subgraph File Format

Saved subgraphs are pretty straightforward. A subgraph is named on a line preceded by the ">" character. Any items found until the next subgraph line are taken to be either nodes, edges, or other subgraphs. Other subgraph names should be preprend by two "_" characters. For example, the following file:

```
>subgraphA
a
b
a-b
__subgraphB
>subgraphB
c
d
c-d
```

describes two subgraphs, subgraphA which contains two nodes (a and b) and one edge (a-b) as well as the nodes and edges defined by subgraphB (c,d, and c-d).

If the subgraph definition file includes an edge or node that is not defined in the database you will see an error.

Valid SQL

Wherever SQL can be used we more or less allow anything that HSQLDB allows. Please see the hsqlSyntax.html file included in the distribution. It should give you a sense of what you can (most things) and can't do.

Edge Definition File

If you want to do things without a database or run just have Zoomgraph make you a temporary database, you can create an edge definition file. Each line is:

```
node1-node2 [node1x,node1y node2x,node2y]
```

See the files `sample` `sampleedgefile_nocoordinates.txt` or `sampleedgefile_withcoordinates.txt` as examples

Zoomable Graphs as Applets

You can run the Zoomable Graph as an applet without the database features. Look at `applet.html` for an example. You can simply paste in the edge definition file into the `INITEDGES` parameter. This is in the same format as the edge definition file above but with a semicolon at the end.