TECHNICAL REPORT NO. 299

GraphView Documentation

Version 1.0

©1989

by

Bjarni Birgisson
Gregory E. Shannon

December 1989

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY
Bloomington, Indiana 47405-4101

GraphView Documentation [†]

Version 1.0

©1989
Bjarni Birgisson
Gregory E. Shannon

Department of Computer Science
Indiana University
Bloomington, Indiana 47405

Technical Report #299

December 1989

0. Introduction

This technical report contains the documentation for GraphView software system for the NeXT computer. For general discussion and motivation of GraphView, see Technical report #295 from the Computer Science Department at Indiana University by Birgisson and Shannon, *GraphView: An Extensible Platform for Manipulating Graphs*. The documentation herein includes:

1. the *GraphView Tutorial* (page 3),
2. the *GraphView User's Manual* (page 11),
3. the specifications of the Objective-C classes used in GraphView (page 21),
4. the specifications of the file format used to store graphs in GraphView (page 55).

This technical report does not contain any documentation on the transformations, generators, or displayers currently available in GraphView. For the up-to-date documentation on these auxiliary programs, contact Greg Shannon at the above address or **shannon@luap.cs.indiana.edu**.

† Development of the GraphView software and its documentation herein has been supported in part by a Hewlett-Packard Faculty Development Fellowship, NSF grant CCR-89-09535, an Indiana University Summer Faculty Fellowship, the Office of Research and Graduate Development at Indiana University, and NSF grant DCR-85-21497.

Copyright © 1989 Trustees of Indiana University, Gregory E. Shannon, and Bjarni Birgisson. All rights reserved. Permission to copy for private use only.

1. Tutorial

Overview

GraphView is a tool for interactively creating, editing, manipulating, displaying, and animating graphs and graph algorithms on the NeXT computer. Graphs can be created directly on the screen, read in from a file, produced by a graph-family generator, or extracted from a database of graphs. Graphs and various characteristics can then be displayed, edited, and transformed using a dynamic external library of transformations written in Objective-C or any other description language. Any of the graphs can be both printed and stored on disk for later retrieval.

GraphView enables the user to work on multiple graphs at once, each appearing in a separate window on the screen. Several display attributes, such as the display of labels or characteristics, are controlled by the user. Subgraphs are copied between different graphs or within the same graph using the standard Copy/Cut/Paste mechanism.

In GraphView, graphs can be arbitrarily manipulated using external programs which communicate with the main program. These transformation programs can be written in any suitable programming language capable of communicating with the main application. In this version, plain ASCII files in a special format are used to transfer information between the programs. Classes of external programs include:

1. Regular transformations which operate on a given input graph, returning one or more graphs as output.
2. Generators, which create new graphs.
3. Displayers, which attempt to lay out a given graph geometrically. <<to be implemented>>

GraphView automatically keeps track of the history of each graph, i.e. which external programs and graphs were used to create it. Some other features described in more detail below include intelligent printing, dynamic display of characteristics, automatic version numbers, a preference panel for setting program defaults, automatic labeling of graphs, and scalable views.

Creating a graph

GraphView can be launched from the Workspace, like any other application, by double-clicking its icon or file name in the browser. It can of course also be started up from a Terminal or Shell window by typing : **G** &.

To create an empty graph, select **New** from the **Windows** menu or press command-n. An empty graph window will be displayed on the screen.

We can now create a graph by adding vertices and edges interactively, using the mouse. To add some vertices, select **Add Vertices** from the **Tools** menu and simply point and click where we want the vertices to appear. To add edges to the graph, select **Add Edge** from the **Tools** menu, point to the source vertex, click and stretch out a line to the destination vertex while holding down the mouse button. When the pointer is inside the destination vertex, release the button, and an edge will be created.

Objects can be moved around by choosing **Point** from the **Tools** menu, selecting the desired objects (by pointing and clicking or dragging out a box) and dragging them to the desired position. Selected objects can also be Cut/Copied and optionally pasted into the same or a different graph.

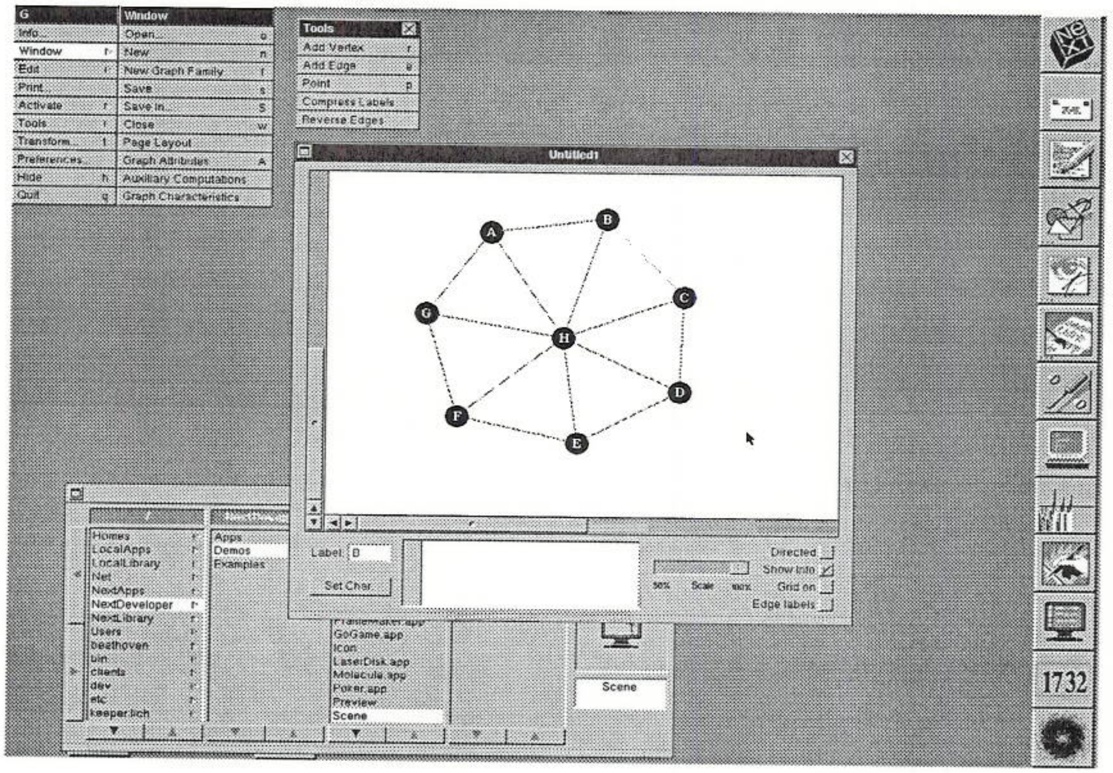


Figure 1. A sample planar graph.

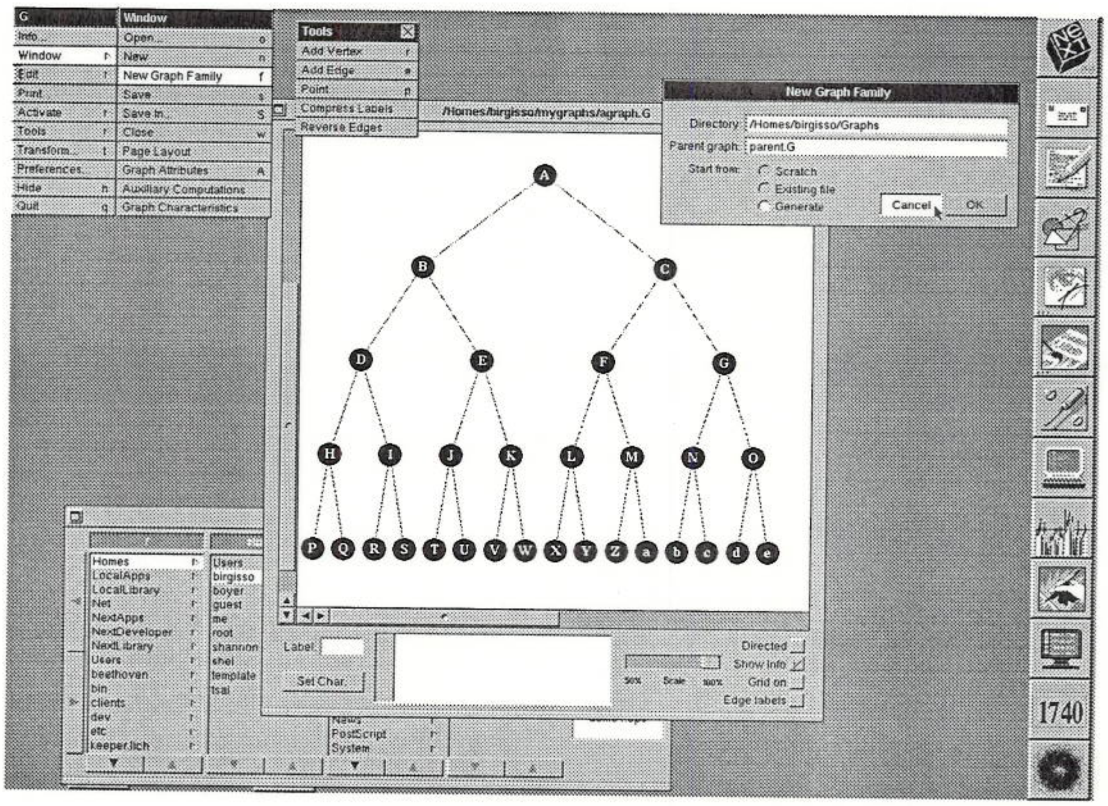
Once a graph has been created as explained above, it can be saved and read back in, using the standard **Save** and **Open** commands in the **Windows** menu.

Starting a new graph family

When trying out a new algorithm, one generally starts out with one or more graphs, and then applies transformations leading to a graph sequence or a family. To start up such a family, select **New Graph Family** from the **Window** menu. To demonstrate how this works, type "~/mygraphs" in the directory field and "agraph.G" for the parent graph. The program will create any necessary directories. From here there are three options on how to start up the sequence.

From scratch. This will open up an empty window in which the parent graph can be created interactively as described above.

Generate. This allows us to select a generator to create some common graphs. For Example select cbt.X from the browser panel, click on **Generate** and type the parameters "5,500,500" (without the quotes) in the parameter field. Cbt.X is a generator to create compete binary trees of a specified height (5 in our case). Click the OK button, after a moment, a window should appear showing a binary tree. The resulting graph is shown in Figure 2 after its window has been enlarged to make the entire graph visible.



Directory: /Homes/birgisso/Graphs

Parent graph: parent.G

Start from: Scratch / Existing file / [✓ Generate]

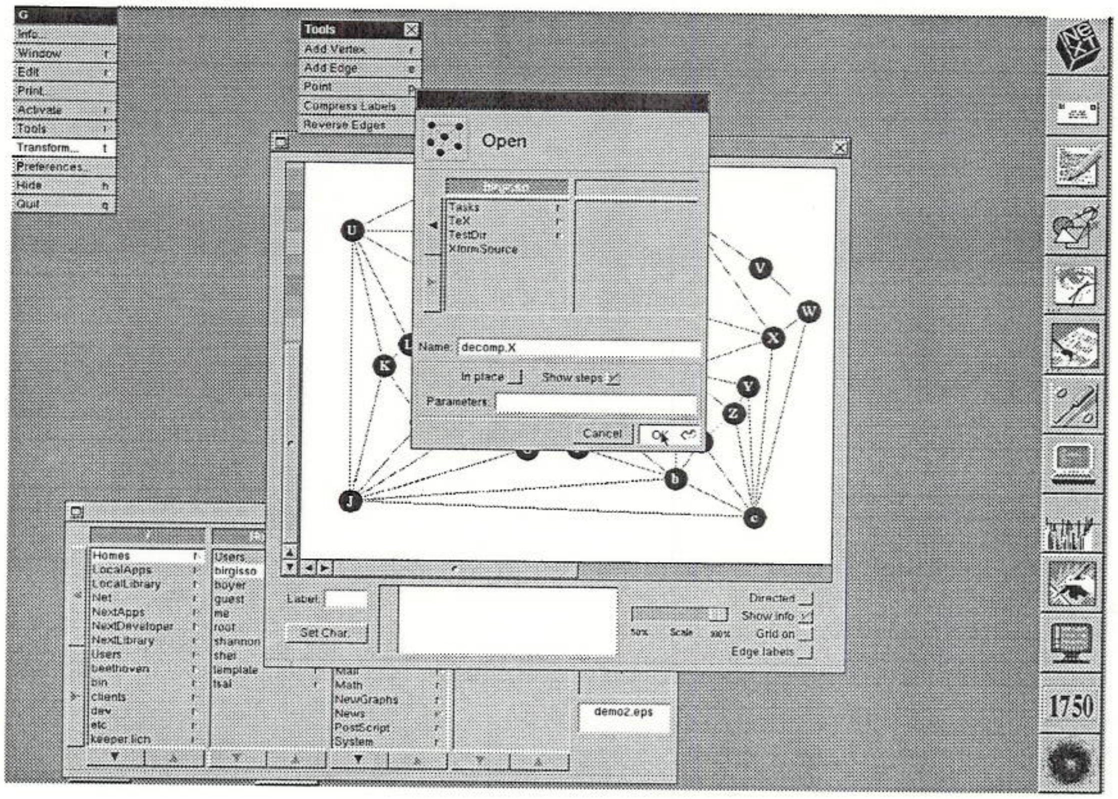
Figure 2. A complete binary tree.

From an existing file. To illustrate this option, go back and bring up the New family window, change the directory name to "~/planar", and the parent graph name to "planar.G". Click the **Existing file** button and then OK. Now the program will show a collection of existing graphs in the default graph directory. Select the file decomp.G and click OK. Now we have a copy of decomp.G, named planar.G in a new directory. If the whole graph isn't visible, adjust the size of the window until it fits. Now we are ready to try out some algorithms.

Applying a transformation.

After starting up the family in the last example, let's use the transformation library to decompose the graph by repeatedly "deleting" vertices of degree 6 or less.

Before selecting the transformation, select **Auxiliary Computations** from the **Window** menu. This brings up a window for messages from the transformation programs.



Name: decomp.X
In place
Show steps ✓
Parameters:

Figure 3. Selecting a transformation.

Now select **Transform** from the main menu. The display should look similar to Figure 3.

The desired transformation is selected by choosing one of the ".X" files in the browser panel. In our case we select decomp.X. The checkboxes can be clicked if we don't want the program to create a new window (in place) and if we want the transformation to show intermediate steps (show steps). Let's click the **show steps** box for this time. If a transformation expects parameters, they are typed into the box at the bottom of the panel. No parameters are expected for decomp.X, so make sure the parameter field is empty.

Press <return> or click the OK button to launch the transformation.

After a while the program will bring up a new window and show the results of the first step of the transformation. It has constructed the first set in the partition, S_1. Vertices in S_1 are marked with S_1 on the display. The panel with the Continue button can be dragged out of the way for a better view of the graph. When ready, click Continue to proceed to the next step. After a few more steps we should have the final result showing the partition by labeling the vertices by S_i indicating the component to which they belong. Figure 4 at the back of this document shows the successive steps.

As another example, let's find a maximal independent set in the graph planar.G. Bring the graph to the front by clicking on its window or selecting it from the **Activate** menu. Select a transformation as before, but now choose p_MIS.X which finds a maximal independent set in a planar graph. P_MIS.X takes no parameters. After showing intermediate steps, the vertices in the MIS are identified by "planar_MIS." The steps in the p_MIS transformation are shown in Figure 5. Note that the input is not shown since it is identical to the first picture in Figure 4 and the final result is not shown either, since it is identical to the last step.

Writing transformations

The current transformation library is written in Objective-C, but could as well be written in any language capable of reading a regular ASCII file and representing the graphs in some way internally. We have created Objective-C classes used to represent the graphs as a collection of objects. Describing them in detail would be lengthy but a short example follows.

We implement a breath-first-search algorithm given in *Data Structures and Algorithms* by Aho, Hopcroft and Ullman (page 243). The original algorithm is as follows:

```
vertex v          -- the source vertex.
queue Q
vertex x y

mark v as visited
enqueue v Q
while Q is not empty
    x <- front Q
   dequeue Q
   for each vertex y adjacent to x
      if y is not marked
         mark y as visited
         enqueue  x Q
         add (x,y) to the bfs-tree
      endif
   endfor
endwhile
```

This could be written as a transformation as follows: (for details see the documentation on the graph classes).

```
// some header code omitted
// v is the source vertex;
Q = [List new];                    // queue for vertices
[Q addObject:[v select]];          // mark v as visited & enqueue
while ([Q count])                  // Q not empty
x = [Q removeObjectAt:0];          // x <- front Q & dequeue
   for (el=[x edgeList], i=[el count]; i--; ) {
      // to get at the adjacent vertices we look at each edge
      e=[el objectAt:i];
      // for directed graphs, ignore incoming edges
      if ([inGraph isDirected] && [e fromVertex] != x)
         continue;
      // get the vertex on the other end of the edge
      y = ([e fromVertex]==x ? [e toVertex] : [e fromVertex]);
      if (![y isSelected]) {          // is y marked as visited?
         [Q addObject:[y select]]; // mark & enqueue y
         [inGraph selectObject:e]; // mark the edge as in
         [e appendInfo:"bfs bfs-tree"];  // the bfs-tree
      } // end if
   } // end for
} // end while
```

This code would be written as single C-function transform(), which then would be linked up with a general main program to handle the communications with the GraphView application. For more detailed examples, see the source code for the transformation libraries.

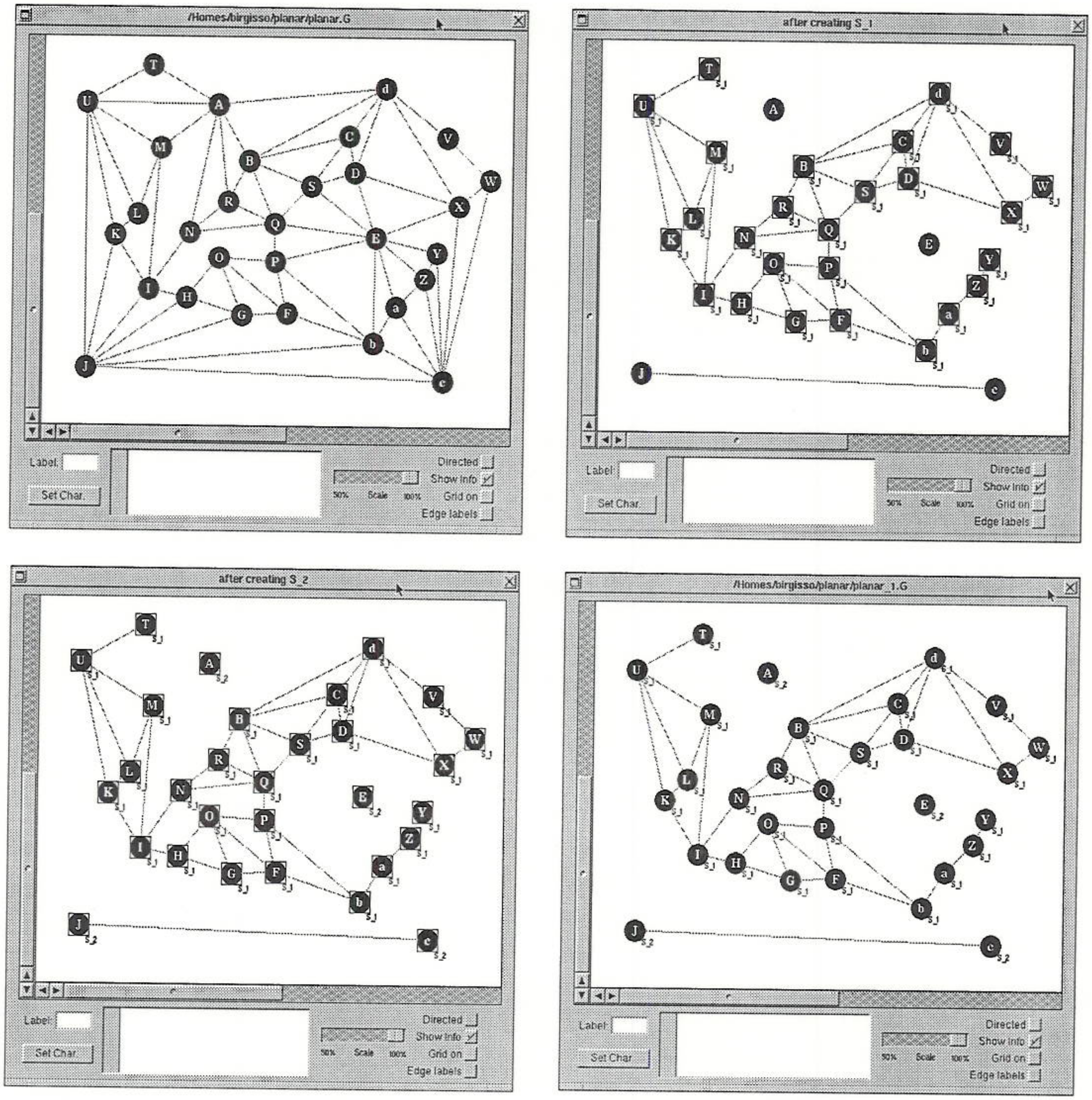


Figure 4. Steps in applying decompose.

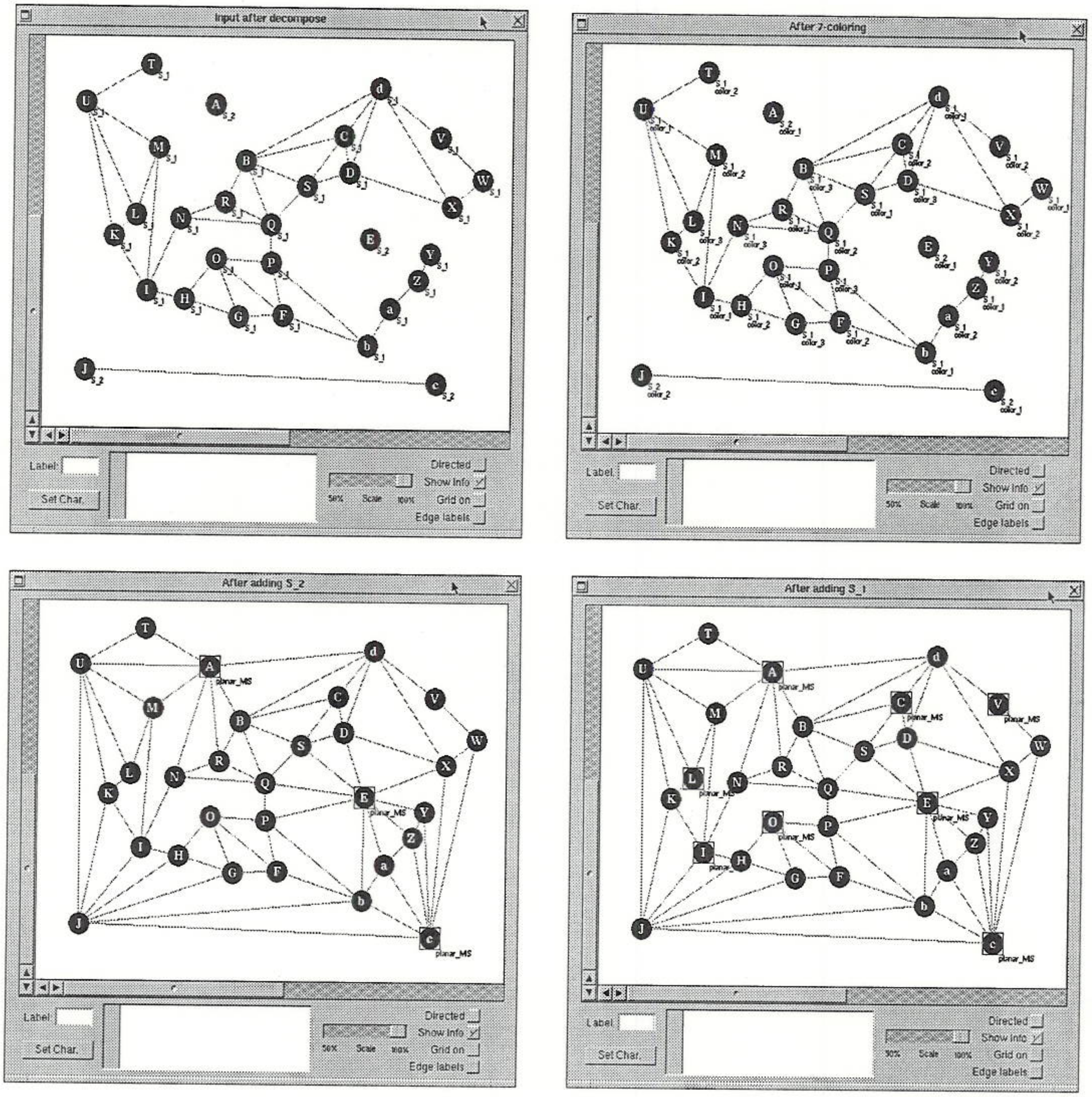


Figure 5. Steps in applying planar_MIS.

2. User's Manual

Starting GraphView

As previously mentioned, GraphView can be launched from the Workspace, like any other application, by double-clicking its icon or file name in the browser. It can of course also be started up from a Terminal or Shell window by typing : **G** &.

Furthermore, the Workspace manager will automatically launch GraphView, when any of it's documents are double-clicked. GraphView documents are expected to have a suffix of ".G" (graphs) or ".X" (transformations).

Since GraphView uses the standard NeXT defaults database, several default values can be specified from the command line or using the **dread/dwrite** commands. However to maintain the integrity of the default data values, it is preferred that these values are changed through the Preferences panel in the program only.

Files and Directories

To make file selection easier, GraphView assumes several default directories and file types. The default directories can all be changed through the Preferences panel. In this version the following directories are used:

- Graph directory. This is the directory where GraphView assumes it will find the graph files. The initial default value is ~/Graphs. If files are selected from another directory using the browser in the Open panel, that directory will become the default directory next time a file is opened or saved using the browser. As mentioned above the default directory can also be changed using the Preferences panel.
- Transformation directory. The initial default value for this directory is ~/Xforms. This is where the transformation programs are expected to be Same rules apply as to the Graph directory above, it can be changed temporarily just by choosing a different directory in the browser, or by using the Preferences panel.
- Generator directory. The initial default value for this directory is ~/Generators. This is where the generator programs are expected to be.
- Displayer directory. <<to be implemented>>

GraphView works with several different file types and uses the following suffixes to distinguish each type. NOTE: Suffixes shorter than 5 characters and using lower case letters are reserved by NeXT. The extensions for GraphView will be registered with NeXT in the near future.

- Graph files. These files have the suffix ".G" and contain the graphs, in an ASCII file format. See Section 4 for a description of the graph file format. The graph files are displayed with the ".G" icon in the Workspace manager.

- History files. These files have the suffix ".H" and contain the history for the graphs, in an ASCII file format. Most graphs have an associated history file with the same name as the graph file and the ".H" suffix. See Section 4 for a description of the history file format. These files are to be maintained by the main Application and/or the transformation programs only. They have no icon.

- Version files. These files have the suffix ".version" and contain only one line of the form: "version xxx\n" Where xxx is the current version number. One version file is kept for each graph family (see below). These files are to be maintained by the main Application and/or the transformation programs only. They have no icon.

- External programs. These files have the suffix ".X" and contain the transformation, generator and displayer programs. These files have an "X" icon when displayed by the Workspace manager.

Graph families are groups of graph files which have all be created by applying transformations to descendants of some common parent. A graph family is started when a graph is created, interactively, using a generator or by other means and placed in a file graph.G for example. This will be the root in the family tree and its descendants are automatically named graph_1.G, graph_2.G and so forth each time a transformation is applied. The version file contains the highest allocated version number for the family and the history file which is kept for each graph, describes the path from the root and the transformations that were applied on the way. See the section on the **New Graph Family** command in the **Window** menu for details on how to start up a new family of graphs.

Graph objects

There are two kinds of objects that make up a graph: vertices and edges. Associated with each object is a (preferably) unique label and any number of lines describing the characteristics of that particular object.

Labels are maintained automatically by GraphView and the user should not have to worry about keeping them unique or in other ways assigning them to objects. Labels can be changed by the user interactively, should the need rise.

Characteristics are usually set by the transformation programs. An example of a characteristic would be the color of a vertex or whether an edge is a part of a particular spanning tree. GraphView provides a convenient mechanism to adjust which characteristics are displayed at any time.

The Graph Window

Each graph appears in a separate window (see below). Several display attributes can be adjusted, both by controls in the window itself and by specifying appropriate defaults through the Preferences panel. An example of a graph window is shown in Figure 1.

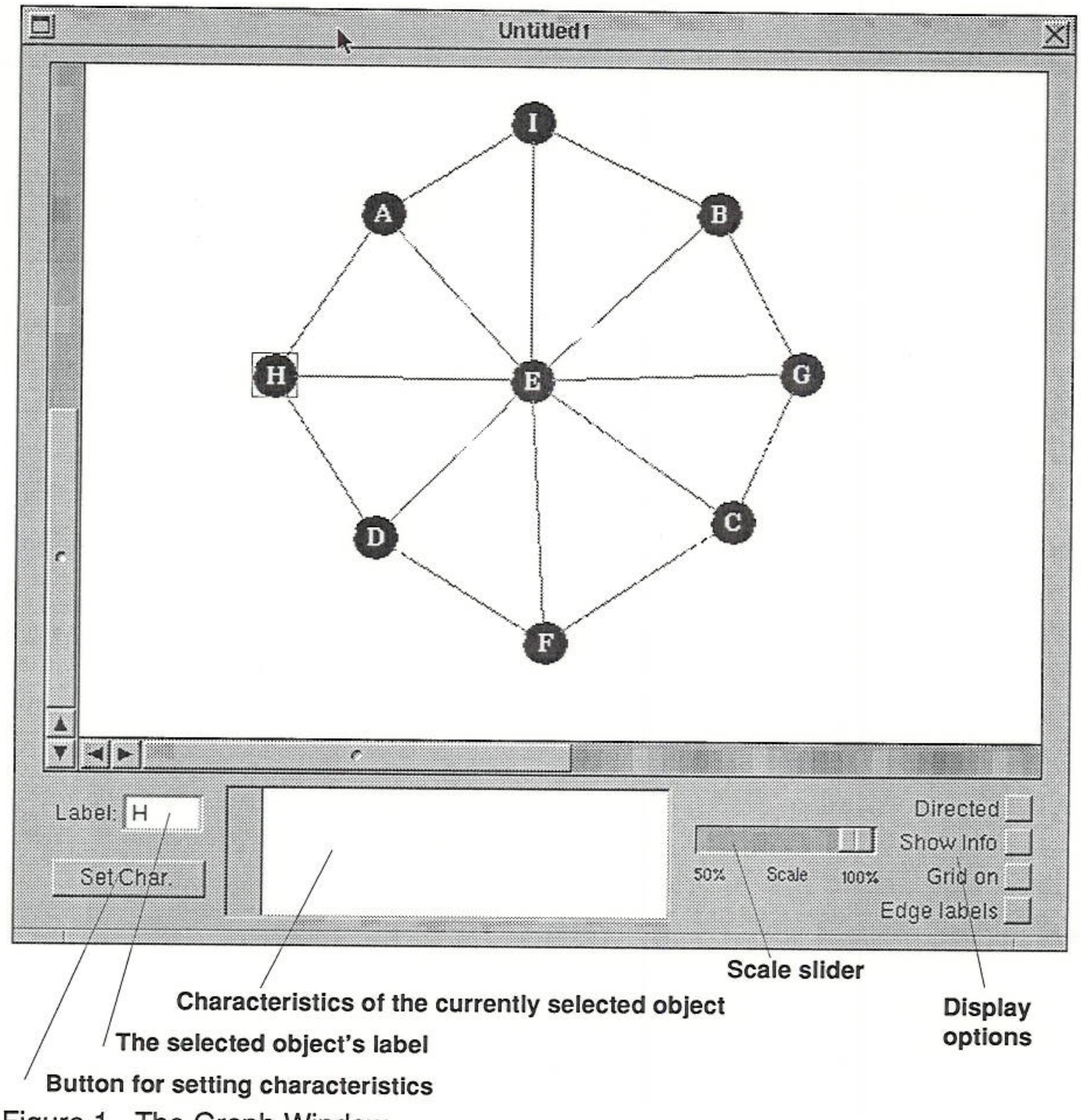

Untitled1
I
A
B
H
E
G
D
C
F
Label: H
Set Char.
50%
Scale
100%
Directed
Show Info
Grid on
Edge labels
Scale slider
Characteristics of the currently selected object
Display options
The selected object's label
Button for setting characteristics

Figure 1. The Graph Window.

Display options: The display option buttons work as follows:

Directed: By clicking this box, the graph is displayed as a directed graph. GraphView remembers the strokes used to draw the edges, and displays the graph accordingly.

Show Info: This box can be clicked to turn on or off the display of characteristics. When this box is checked, the display information in the Preferences panel determines which characteristics are actually displayed. NOTE: Having this box checked slows down the display speed considerably for large graphs.

Grid on: When this box is checked, a rectangular grid overlays the image. The squares are 40 by 40 points or approximately a square inch.

Edge labels: Checking this box causes labels to be displayed for the edges as well as for the vertices in the graph. NOTE: This option slows down display speed somewhat.

Scale slider: By adjusting the scale slider, the graph can be scaled down to 50% of its default size, thereby bringing more of the graph into view.

Labels: In the Label field, GraphView displays the label of the last selected object (see below). It can be changed, simply by typing in the desired label and pressing return in the field, while the object is still selected.

Characteristics: This field displays the characteristics associated with the last selected object. To change the characteristics, type in the new information and press the **Set Char.** button while the object is still selected. The program assumes characteristics are any number of lines of the following form:

```
keyword  some_text_to_be_displayed
```

An example of characteristics for a vertex with color blue, belonging to a connected component number 3 would be:

```
color blue connected-comp  cc-1
```

Then setting characteristics displayed for vertices in this graph to:

```
color
connected-comp
```

would display

```
blue
cc-1
```

somewhere close to the vertex in the graph. In theory, any number of characteristics can be displayed at one time and the strings can be of any length. Keep in mind though that the display quickly becomes cluttered when too much is displayed.

Selecting Objects - Cut/Copy/Paste

Objects in the graph are selected simply by clicking on them or dragging out a box around the desired objects. The usual shift-click extensions apply for adding to the current selection. All the objects can be selected using the **Select All** option in the **Edit** menu. NOTE: Selection can only be performed when in **Point** mode, see the section on the **Tools** menu for details.

Once objects have been selected they can be moved by clicking and dragging them to the new position, or cut and/or copied using the standard operations from the **Edit** menu and then pasted into any graph. GraphView automatically assigns new unique labels, in the same internal order, when pasting a subgraph.

Due to the special characteristics of graphs, the following should be noted about copying, cutting and pasting. An edge can only be copied and pasted if both of it's adjacent vertices are copied/pasted with it. When a vertex is cut from a graph, all its adjacent edges will be cut also, whether they are selected or not.

The Main Menu

GraphView's main menu contains the commands described below. Many of these commands display submenus related to specific areas of functionality. These submenus and the commands they contain are described in the sections that follow.

Info
The Info command displays information about GraphView, including the version number and copyright information.

Window
The Window command displays a menu that contains commands for performing operations on graph windows and activating several additional windows. These commands include opening and closing windows, and saving changes made to graph windows.

Edit
The Edit command displays a menu that contains the standard **Cut/Copy** and **Paste** commands.

Print
The Print command displays the standard Print panel, which can then be used to print the contents of the currently active window.

Activate
This command displays a menu of the open graph windows. A window chosen from this list moves to the front and becomes the currently active window.

Tools

This command brings up a submenu with several commands that can be applied to the currently active window.

Transform

The **Transform** command is used to invoke an external transformation. All the available transformations are expected to be located in the default transformation directory (see **Files and Directories** above). All transformations operate on the currently active graph (the one in the top window). To launch a transformation, select it from the browser panel just as any other file, type in the necessary parameters and hit return or click the OK button. There are two options available when starting a transformation. By checking the boxes labeled **in place** and **show steps** you can tell GraphView to run the transformation in the current window and show intermediate results respectively. For details on how to write transformation programs, see appendix B.

Preferences

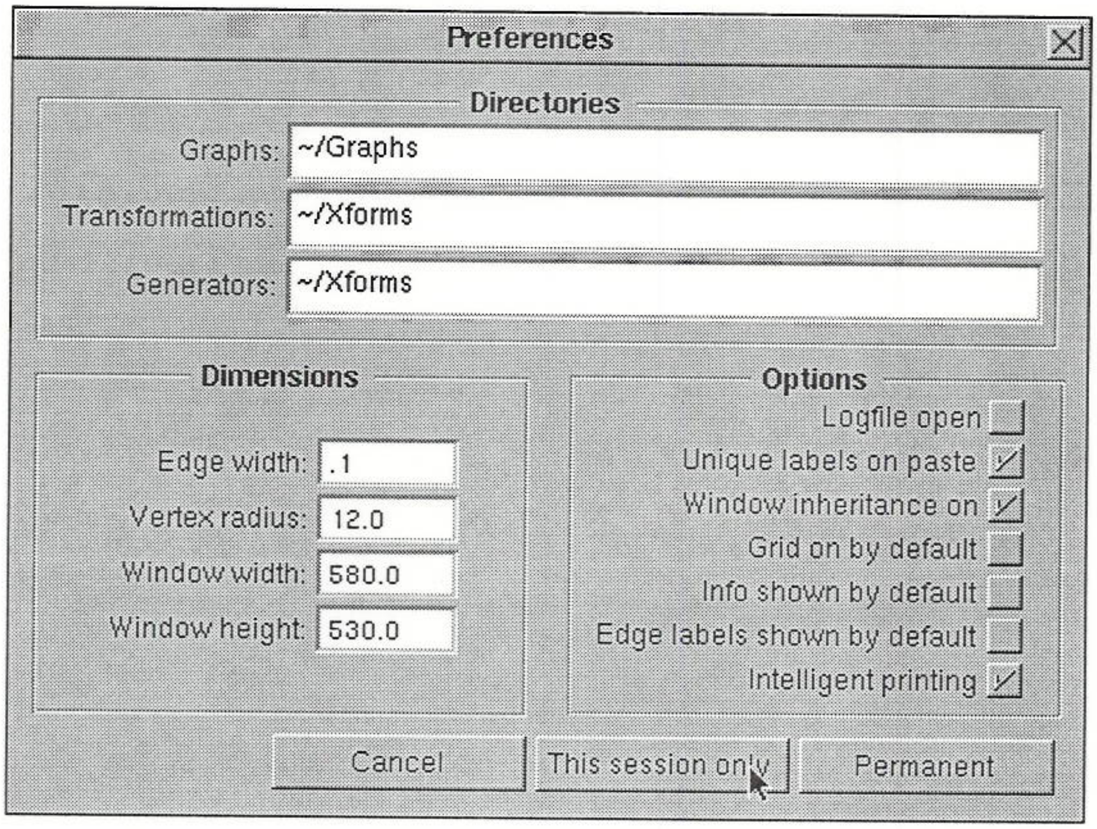
This command displays a panel from which the programs default values can be changed, either permanently or for a single session. Most of the options are self-explained. Only a few are mentioned here.

Logfile open: This option decides whether the contents of the Auxiliary Computations window are saved to a file. If it is on, a file named MMDD-HHMM.log, where MM, DD, HH and MM are month, day, hour and minute respectively, is created in the Graph directory and the contents of the Auxiliary Computations window are echoed to that file.

Unique labels on paste: When this box is checked GraphView makes sure that labels are kept unique on a paste. In other words, it allocates new labels to the just-pasted subgraph.

Window inheritance on: If this box is checked, the default display attributes are overridden, such that a transformed graph will have exactly the same attributes as its parent.

Intelligent printing: Choosing this option, allows GraphView to decide the orientation (portrait/landscape) of printed graphs, depending on the geometry of each graph, thereby overriding the setting in the Page Layout window.


Preferences

Directories

Graphs: ~/Graphs

Transformations: ~/Xforms

Generators: ~/Xforms

Dimensions

Edge width: .1

Vertex radius: 12.0

Window width: 580.0

Window height: 530.0

Options

Logfile open [ ]

Unique labels on paste [✓]

Window inheritance on [✓]

Grid on by default [ ]

Info shown by default [ ]

Edge labels shown by default [ ]

Intelligent printing [✓]

Figure 2. The Preferences Panel.

Hide
The Hide command hides the GraphView program. Windows and menus temporarily disappear from the Workspace. To activate the program, double-click its icon and the windows and menus reappear in their previous locations.

Quit
The Quit command stops the program. NOTE: it does not check to see if all files have been saved.

The Window Menu

Open

The Open command prompts for a graph file, reads it in and displays it in a graph window. The file is specified by typing the file name or path name in the text field of the panel that appears when this command is selected. Alternatively, one or more files can be selected (using the standard shift-click) extension if necessary and opened by double-clicking any of them. Another way to open a graph file is to simply double-click it in the Workspace browser. The Workspace manager will send the necessary messages to Graphview and launch it if needed.

New

The New command opens an empty graph window titled "Untitledxx", where xx is a running number. This window can be used to create a new graph. When the file is saved, GraphView prompts for a file name.

New Graph Family

When this command is selected, a panel is displayed, allowing you to specify the directory that is to contain the new family and the name of the parent graph. GraphView will create all necessary directories if they don't exist. The parent graph can then be created in one of three ways:

1. From scratch. An empty window is displayed. Design the parent graph interactively, as described above.
2. From an existing file. Select a file form an open panel to start the family.
3. Generate. This will invoke an external generator program to generate the parent graph. The generator to be used is selected from a browser panel just like a transformation (see **Transform** above).

Save

The Save command writes the contents of the current window to a file. If the window is titled "Untitledxx", a panel appears asking for a name for the new file. After the file is saved, it remains in the window.

Save As

The Save As command saves the contents of a window you've created with the New command (see above), or saves an existing graph under a new name or to a new directory. When you choose Save As, a panel appears asking you to enter the new name for the graph. Type the new file name (or a complete path name, if you want to specify a different directory), and press Return. After you use the Save As command, the contents of the window are associated with the new graph name. If you use the Save As command to save an existing graph under a new name, the old version continues to exist on the disk under its old name.

Close

Closes the currently active window. If this is a graph window, GraphView does not check whether any unsaved changes will be lost.

Page Layout
The Page Layout command displays the standard Page Layout panel, which lets you choose among various paper sizes, scaling factors, and orientations for text printed from the main window. NOTE: The chosen setting for Landscape/Portrait printing does only affect graphs printed with the intelligent printing option off. See the Preferences panel for details.

Graph Attributes
This command brings up a panel showing the file name and the directory of the current graph, as well as its history and characteristics displayed for its objects. The history can be cleared and the version numbers reset by clicking the appropriate button. To change the characteristics displayed for the graph's objects, simply type in the keywords identifying the characteristics, e.g. "color," separated by spaces newlines or tabs and click the **Set** button.

Auxiliary Computations.
Selecting this command brings up a window which the transformation programs can write to. things are set up in such a way that by simply writing to stdout or stderr, and doing an fflush afterwards, the transformation programs can send informative messages back to GraphView. These messages are displayed in the auxiliary computations window. If the logfile option is on in the Preferences panel, the messages are also echoed to a logfile. The window can be emptied by clicking the **Empty window** button.

Graph Characteristics
Each graph can have certain characteristics associated with it. This can be any amount of text which is appended to a text field in a special window when a graph is opened. This command activates the window displaying these characteristics. Graph characteristics are currently set and modified by the transformation programs only. they cannot be changed or set directly through GraphView. The window can be emptied by clicking the **Empty window** button.

The Edit Menu

The Edit menu provides the standard editing commands **Cut**, **Copy**, **Paste**, and **Select All**.

Cut, Copy, Paste
These commands let you delete, copy, or move objects, either within a window or between windows. Each of these commands applies to the key window.

The Cut and Copy commands place a copy of the selected objects onto the pasteboard. From the pasteboard, the objects can be repeatedly pasted with the Paste command. The pasteboard holds only one selection at a time; each new Cut or Copy operation overwrites the previous contents of the pasteboard.

NOTE: When working in a text field, for example when editing the characteristics of a selected object, these commands apply to the text in that field rather than the graph in the window. By clicking the graph, it will again become the recipient of the messages sent by these commands.

Select All
The Select All command selects all objects in the currently active window.

The Tools Menu

The Tools menu contains several commands used when creating or editing a graph interactively.

Add Vertex
When editing graphs interactively, GraphView has three basic modes. **Add Vertex** and the next two commands, **Add Edge**, and **Point** switch between modes. After selecting this command, vertices can be added to the graph by pointing to the desired position and clicking. It is possible to move the vertex around to a new position while holding down the mouse button, since the vertex isn't created until the button is released. GraphView automatically assigns a new label to the vertex but the characteristic field is initially blank.

Add Edge
This command allows edges to be added to a graph. To add an edge, click on the source vertex and while holding down the mouse button, drag towards the destination vertex, releasing the button with the mouse pointer within the radius of the destination vertex. GraphView assigns labels to edges automatically.

Point
This command allows objects in the graph to be selected. GraphView has to be in this mode for selection to work. See the section Selecting Objects for details.

Compress Labels
After deleting vertices and edges, the label sequence in the graph will have unused gaps. This command compresses the label sequence for both edges and vertices, leaving no gaps and still maintaining the internal ordering of the labels.

Reverse Edges
This command only has effect in directed graphs. It allows the direction of all *selected* edges to be reversed.

3. Objective C Class Specifications

This section contains the class specifications for GraphView: Vertex, page 21; Edge, page 25; Graph, page 31; GraphObject, page 39; GraphView, page 43.

3.5. Vertex

INHERITS FROM	GraphObject :Object
REQUIRES HEADER FILES	Vertex.h, <objc/List.h>
DEFINED IN	GraphView

CLASS DESCRIPTION

Vertex objects are one of the two main classes composing a graph. Associated with each vertex is a list of edge objects and a coordinate, representing its geometrical position in the graph. The edgelist contains both incoming and outgoing edges. Only one copy is kept of a single edge, which means that each edge occurs in exactly two lists, the one of its source vertex and its destination vertex. Care must be taken when modifying the edgelists directly to properly update both lists.
NOTE: Although a graph may be undirected, it is stored in the same way as a directed graph. To find whether an edge is an outgoing or incoming edge, the result of the **fromVertex** method of the edge class can be compared to the vertex in question.

INSTANCE VARIABLES

Inherited from Object	struct _SHARED	*isa;
Inherited from GraphObject	int	seq1;
	int	seq2;
	BOOL	selected;
	int	charlen;
	char	*c;
	char	label[3];
Declared in Vertex	id	edgelist;
	NXPoint	pos;

edgelist	The list of edges. See the **List** class for details.
pos	The geometrical position of the vertex (center) in the graph.

METHOD TYPES

Creating and freeing a Vertex	+ newVertexAt:label:info:edges: - remove
Writing the Vertex to a file	- writeVertexTo:
Managing the edgelist	- sortEdgeList - edgeList - getEdgeTo:
Geometric methods	- location - bounds: - isHit: - moveTo: - moveBy:
Drawing - drawInRect:	
Querying the object type	- type

FACTORY METHODS

newVertexAt:label:info:edges:

+ **newVertexAt:**(NXPoint *)*p* **label:**(char *)*l* **info:**(char *)*c* **edges:**(int)*ec*

Creates a new instance of Vertex, positioned at *p*, with label *l* and characteristics *c* (may be NULL). If an estimate is known of how many edges will be connected to the vertex, that number should be given as *ec*; if this is not known *ec* should be 0.

INSTANCE METHODS

bounds

– (NXRect *)**bounds**

Returns a pointer to an NXRect structure containing a rectangle that encloses the vertex.

drawInRect:

– **drawInRect:** (NXRect *)*aRect*

Checks if the bounds of the receiver intersect *aRect* if so the vertex is drawn along with its outgoing edges. Only outgoing edges are drawn to prevent the same edge from being drawn twice. If *aRect* is NULL the vertex is drawn without checking. <<There is no method for drawing only the vertex, without the edges. It can easily be added if needed>>

edgeList

– **edgeList**

Returns the List object containing the connected edges.

getEdgeTo:

– **getEdgeto:***vertex*

This method searches the edgelist for an edge between the receiver and *vertex.* If no edge exists, **nil** is returned. This method only looks for an edge from the receiver to *vertex*—not the other way.

isHit:

– (BOOL)**isHit:**(NXPoint *)*aPoint*

Returns YES if *aPoint* hits the vertex, NO otherwise.

location

– (NXPoint *)**location**

Returns a pointer to an NXPoint structure containing the coordinate of the center of the vertex.

moveBy:

– **moveBy:**(NXCoord)*dx* **:**(NXCoord)*dy*

Adds *dx* and *dy* to the x and y coordinates of the vertex .

See also: **moveTo:**

moveTo:

– **moveTo:**(NXPoint *)*aPoint*

Sets the position of the vertex to *aPoint*.

See also: **moveBy::**

remove

– **remove**

Frees all storage allocated to the vertex and all its edges. Also removes the edges from the edgelists of the adjacent vertices. Note that the vertex is not removed from the vertex list of the graph. See the **Graph** class for details.

sortEdgeList

– **sortEdgeList**

Sorts the edgelist so that it reflects clockwise ordering of the edges around the vertex. Note that the edgelist is not necessarily kept in sorted order at all times.

type

– **type**

Returns the constant G_VERTEX.

writeVertexTo:

–**writeVertexTo:**(NXStream *)*stream*

Writes the information associated with the vertex to the stream *stream* in the format required by .G files.

See also: The I/O methods for the Graph and Edge classes.

3.1. Edge

INHERITS FROM	GraphObject : Object
REQUIRES HEADER FILES	Edge.h
DEFINED IN	GraphView

CLASS DESCRIPTION

Edges are one of the two main classes making up a graph. An edge consists of an ordered pair of vertices, the source and destination vertex and a reference to the graph containing it. Edges are stored the same way in directed and undirected graphs, and the direction of the vertex is determined by the way the user enters it in the graph. (This can be changed using the **reverse** method in directed graphs.)

INSTANCE VARIABLES

Inherited from Object	struct _SHARED	*isa;
Inherited from GraphObject	int	seq1;
	int	seq2;
	BOOL	selected;
	int	charlen;
	char	*c;
	char	label[3];
Declared in Edge	id	graph;
	id	fromv;
	id	tov;

graph	The graph containing the edge. It is necessary for the edge to find out whether the graph is directed or not when drawing.
fromv,tov	The source and destination vertex respectively.

Class Specifications: *Edge*

METHOD TYPES

Creating and freeing an Edge	+ newEdgeFrom:to:inGraph: label:info:
	- remove
Writing the Edge to a file	- writeEdgeto:
	- writeEdgefrom
Accessing the vertices	- fromVertex
	- toVertex
	- setFromVertex:
	- setToVertex:
	- isFrom:to:
Geometric methods	- bounds:
	- isHit:
	- intersectsRect:
	- angle
	- angleWithRespectTo:
Drawing	- drawInRect:
Reversing the edge	- reverse
Querying the object type	- type

FACTORY METHODS

newEdgeFrom:to:inGraph:label:info:edges:

+ **newEdgeFrom:***source* **to:***destination* **inGraph:***graph*
  label:(char *)*l* **info:**(char *)*c*

Creates a new instance of Edge, from *source* to *destination*, with label *l* and characteristics *c* (may be NULL). The edge is in *graph*.

INSTANCE METHODS

angle

– (float)**angle**

Returns the angle in degrees with respect to the source vertex. Angle 0 is at "3 o'clock" when looking at the vertex on the screen. This is a cover method for **angleWithRespectTo:**.

See also: **angleWithRespectTo:**

angleWithRespectTo:

– (float)**angleWithRespectTo:***vertex*

Returns the angle in degrees with respect to the given vertex; 0 degrees is 3 o'clock. *Vertex* must be the source or destination edge of the receiver.

See also: **angle**

bounds

– (NXRect *)**bounds**

Returns a pointer to an NXRect structure containing a rectangle that encloses the edge.

drawInRect:

– **drawInRect:** (NXRect *)*aRect*

Checks if the edge intersects *aRect,* if so the edge is drawn. If *aRect* is NULL the vertex is drawn without checking.

fromVertex

– **fromVertex**

Returns the source vertex.

See also: **toVertex, setFromVertex:, setToVertex:**

intersectsRect:

– (BOOL)**intersectsRect:**(NXRect *)*aRect*

Returns YES if the edge intersects *aRect*, NO otherwise.

isFrom:to:

– (BOOL)**isFrom:***svertex* **to:***dvertex*

Returns YES if the edge is from *svertex* to *dvertex*, NO otherwise.

isHit:

– (BOOL)**isHit:**(NXPoint *)*aPoint*

Returns YES if *aPoint* hits the edge or is "close" to it, NO otherwise. For now this method is implemented using the PostScript **instroke** operator to handle curved edges properly. <<this is too slow for large graphs and will be changed>>

remove

– **remove**

Frees all storage allocated to the edge and removes it from the edgelists of both vertices.

See also: **remove** (Vertex)

reverse

– **reverse**

Swaps the source and destination vertices.

setFromVertex:

–**setFromVertex:***v*

Sets the source vertex to *v*.

See also: **toVertex, fromVertex:, setToVertex:**

setToVertex:

–**setToVertex:***v*

Sets the destination vertex to *v*.

See also: **toVertex, fromVertex:, setFromVertex:**

toVertex

– **toVertex**

Returns the destination vertex.

See also: **fromVertex, setFromVertex:, setToVertex:**

type

– **type**

Returns the constant G_EDGE.

writeEdgefrom:

–**writeEdgefrom:**(NXStream *)*stream*

Writes the information associated with the edge to the stream *stream* in the format required by the edgefrom line in .G files.

See also: **writeEdgeto:** and the I/O methods for the Graph and Vertex classes.

writeEdgeto:

–**writeEdgeto:**(NXStream *)*stream*

Writes the information associated with the edge to the stream *stream* in the format required by the edgeto line in .G files.

See also: **writeEdgefrom:** and the I/O methods for the Graph and Vertex classes.

3.2. Graph

INHERITS FROM	Object
REQUIRES HEADER FILES	Graph.h
DEFINED IN	GraphView

CLASS DESCRIPTION

A graph is implemented as a list of vertices which each has a list of edges. See the description for those classes for details. Besides the vertex list, the graph contains two lists of selected objects, one for vertices and one for edges. It also stores information about the graph type (directed or undirected) and the characteristics associated with the graph.

INSTANCE VARIABLES

Inherited from Object	struct _SHARED	*isa;
Declared in Graph	BOOL	showInfo;
	int	Gtype;
	id	vList;
	id	sele;
	id	selv;
	id	infoObject;
	id	charEdges;
	id	charVertices;
	char	Elabel[2];
	char	Vlabel[2];

showInfo	Records whether extra information such as edge labels are to be shown when drawing the graph. the default is NO.
Gtype	Records the graph type 1 means directed 0 undirected. The constants G_DIRECTED and G_UNDIRECTED should be used when referring to the graph type.
vList	The vertex list for the graph. It is not kept in any particular order.
sele,selv	Lists of selected edges and vertices respectively.

infoObject	A GraphObject (see its class description for details) used only to keep the characteristics associated with the graph.
charEdges,charVertices	GraphObjects used to store the characteristics displayed for the edges and vertices of the graph respectively.
Elabel,Vlabel	Contain the highest label allocated to edges and vertices in the graph. Pasting from other graphs does not affect these variables but reading from files does.

METHOD TYPES

Creating and freeing a Graph	+ newGtype: - free
I/O methods	- readGraphFromFile: - writeGraphToFile:
Managing characteristics	- setGraphInfo: - getGraphInfo
Managing the display of characteristics	- setDispCharEdges: - setDispCharVertices: - getDispCharEdges - getDispCharVertices
Adding objects	- addVertex: - addEdgeFrom:to: - insertVertex: - insertEdge:
Accessing objects	- getVertexAt: - getEdgeAt: - vertexList
Managing the state	- setDirected: - isDirected - setShowInfo: - isInfoShown
Managing the selection	- selectedVertices - selectedEdges

- unselectAll
- unselectObject:
- selectObject:
- getSelectedBounds:
- addToSelection:
- removeSelection
- readSelectionFrom:andGenerate:
- writeSelectionTo:

FACTORY METHODS

newGtype:

+ **newGtype:**(int)*type*

Creates a new instance of graph, with Gtype set to *type.* ShowInfo is set to NO. The selection lists are empty and the graph has no characteristics.

INSTANCE METHODS

addEdgeFrom:to:

– **addEdgeFrom:***svertex* **to:***dvertex*

Creates a new instance of edge form *svertex* to *dvertex* with the next unallocated label and no characteristics.

See also: **insertEdge:**

addToSelection:

– **addToSelection:**(NXRect *)*aRect*

Adds to the current selection all objects, edges and/or vertices which intersect *aRect.*

addVertex:

– **addVertex:** (NXPoint *)*aPoint*

Creates a new instance of vertex at *aPoint* with the next unallocated label and no characteristics.

See also: **insertVertex:**

free

– **free**

Releases all storage allocated to the graph and its objects.

getDispCharEdges

– (char *)**getDispCharEdges**

Returns pointer to a string of characteristics to be displayed for the graph's edges. This will be a string of space-, tab-, or newline-separated keywords.

See also: **setDispCharVertices:, setDispCharEdges:, getDispCharVertices.**

getDispCharVertices

– (char *)**getDispCharVertices**

Returns pointer to a string of characteristics to be displayed for the graph's edges. This will be a string of space, tab or newline separated keywords.

See also: **setDispCharVertices:, setDispCharEdges:, getDispCharEdges.**

getEdgeAt:

– **getEdgeAt:** (NXPoint *)*aPoint*

Searches the edgelists of all the vertices for an edge that is hit by *aPoint* and returns it if found. Otherwise returns **nil**.

getGraphInfo

– (char *)**getGraphInfo**

Returns a pointer to the characteristics associates with the graph (may be NULL).

See also: **setGraphInfo:, getInfo** (GraphObject).

getVertexAt:

– **getVertexAt:** (NXPoint *)*aPoint*

Searches for a vertex hit by *aPoint* and returns it if found. **Nil** otherwise.

getSelectedBounds:

– (BOOL)**getSelectedBounds:**(NXRect *)*aRect*

Calculates a rectangle enclosing all the selected objects in the graph and puts it in the structure pointed to by *aRect.* Returns YES if the rectangle is non-empty and NO otherwise.

insertVertex:

– **insertVertex:***vertex*

Inserts the already created object *vertex* into the graph.

See also: **addVertex:**

isDirected

– (BOOL)**isDirected**

Returns YES if the graph is directed, NO otherwise.

See also: **setDirected:**

isInfoShown

– (BOOL)**isInfoShown**

Returns YES if the showInfo flag is set, NO otherwise.

See also: **setShowInfo:**

readGraphFromFile:

– (BOOL)**readGraphFromFile:**(NXStream *)*stream*

Reads the graph from an already opened file *stream*. The graph has to be created with **newGtype:** and must be empty before this method can be applied. << This could be an easy way to create the union of two graphs .>>

See also: **writeGraphToFile:.**

readSelectionFrom:

– (BOOL) **readSelectionFrom:**(NXStream *)*stream*
andGenerate:(BOOL)*flag*

Reads in a description of vertices and edges and adds to the graph. The objects read are made the currently selected objects. Used to read from the pasteboard as a response to a paste operation. If *flag* is YES new labels are generated for the objects read and the Elabel and Vlabel variables updated accordingly.
NOTE: The file format required for this method is the same as used by **writeSelectionTo:** but a bit different than the format for .G files to speed up cutting and pasting.

See also: **writeSelectionTo:**

removeSelection

– **removeSelection**

Removes all selected objects from the graph and frees that storage.

selectedEdges

– **selectedEdges**

Returns the list of selected edges.

See also: **selectedVertices**

selectedVertices

– **selectedVertices**

Returns the list of selected vertices.

See also: **selectedEdges**

selectObject:

– **selectObject:***o*

Adds the object *o*, which can either be a vertex or an edge to the current selection and marks it as selected.

See also: **unselectObject:, unselectAll**

setDirected:

– **setDirected:**(BOOL)*flag*

Sets the graph type to directed if *flag* is YES or undirected if *flag* is NO.

See also: **isDirected**

setDispCharEdges:

– **setDispCharEdges:**(char *)*aString*

Sets *aString* as the string that determines which characteristics are displayed for the edges of the graph.

See also: **setDispCharVertices:, getDispCharEdges, getDispCharVertices.**

setDispCharVertices:

– **getDispCharVertices:**(char *)*aString*

Sets *aString* as the string that determines which characteristics are displayed for the vertices of the graph.

See also: **setDispCharEdges:, getDispCharEdges, getDispCharVertices.**

setGraphInfo:

– **setGraphInfo:**(char *)*characteristics*

Associates *characteristics* with the graph.

See also: **getGraphInfo, setInfo** (GraphObject)

setShowInfo:

– **setShowInfo:**(BOOL)*flag*

Sets the showInfo flag to *flag* which should be YES or NO.

See also: **isInfoShown**

unselectAll

– **unselectAll**

Empties the current selection and marks the objects as unselected but does not remove them from the graph.

See also: **selectObject:, unselectObject:**

unselectObject:

– **unselectObject:***o*

Removes the object *o*, which can either be a vertex or an edge from the current selection and marks it as unselected.

See also: **selectObject:, unselectAll**

vertexList

– **vertexList**

Returns the list of vertices.

writeSelectionTo:

– **writeSelectionTo:**(NXStream *)*stream*

Writes the current selection to the file *stream* in a format suitable for **readSelectionFrom:** This format is different for the one used by .G files and is used for writes to the pasteboard as a response to cut and/or copy requests. The edgelists are not sorted before the write is performed.

See also: **readSelectionFrom:**

writeGraphToFile:

–(BOOL)**writeGraphToFile:**(NXStream *)*stream*

Writes the graph to an already open file *stream* in a .G file format which can be read by **readGraphFromFile:**. This method guarantees that the edgelists will be sorted, reflecting a counterclockwise ordering around the edges in the file and after the method is applied. The graph is not freed. YES is returned if the method succeeds and NO if an I/O error occurred.

See also: **readGraphFromFile:, sortEdgeList** (Vertex).

3.3. GraphObject

INHERITS FROM	Object
REQUIRES HEADER FILES	GraphObject.h
DEFINED IN	GraphView

CLASS DESCRIPTION

GraphObjects are used as a common class for the implementation of both edges and vertices in a graph. They provide features common to both types of graph objects, such as selection, labels, characteristics etc.

INSTANCE VARIABLES

Inherited from Object	struct _SHARED	*isa;
Declared in GraphObject	int	seq1;
	int	seq2;
	BOOL	selected;
	int	charlen;
	char	*c;
	char	label[3];

seq1,seq2,seq3	These are temporary variables which can be used to store a particular state of the object. Seq1 and seq2 are used when saving and reading a graph to/from a file to store information about connectivity.
selected	The current state of the selection.
charlen	Length of the characteristics associated with the object.
c	A pointer to the characteristics. May be NULL if no characteristics are currently associated with the object.
label	A string containing the object's label.

METHOD TYPES

Creating and freeing a GraphObject	+ new - free
Managing the Selection	- select - unselect - isSelected
Managing the temp. variables	- setSeq1: - setSeq2: - setSeq3: - getSeq1 - getSeq2 - getSeq3
Handling labels	- setLabel: - getLabel
Handling characteristics	- appendInfo: - setInfo: - getInfo
Querying the object type	- type

FACTORY METHODS

new

+ **new**

Creates a new instance of GraphObject. The object is unselected, the label is "" and c and charlen are NULL and 0 respectively.

INSTANCE METHODS

appendInfo:

– **appendInfo:**(char *)*aLine*

Appends *aLine* to the characteristics already associated with the object. Storage is allocated and the string copied. The charlen variable is reset.

See also: **getInfo, setInfo:**

free

– **free**

Releases all storage allocated for the object.

getInfo

– (char *)**getInfo**

Returns a pointer to characteristics for the object, pointer maybe be NULL.

See also: **setInfo:, appendInfo:**

getLabel

– (char *)**getLabel**

Returns a pointer to the label associated with the object.

See also: **setLabel:**

getSeq1, getSeq2, getSeq3

– (int)**getSeq1**

Returns the temporary state variable seq1. Similar for seq2 and seq3.

See also: **setSeq1:**

isSelected

– (BOOL)**isSelected**

Returns YES if the object has been selected, NO otherwise.

See also: **select, unselect**

select

– **select**

Marks the object as selected.

See also: **isSelected, unselect**

setInfo:

– **setInfo:**(char *)*characteristics*

Associates *characteristics* with the object. Storage is allocated and the string copied. The charlen variable is set to reflect the new length.

See also: **getInfo, appendInfo:**

setLabel:

– **setInfo:**(char *)*label*

Sets the objects label to *label.*

See also: **getLabel**

setSeq1:, setSeq2:, setSeq3:

– **setSeq1:**(int) *i*

Sets the corresponding temporary variable to *i.*

See also: **getSeq1**

unselect

– **unselect**

Marks the object as unselected.

See also: **isSelected, select**

type

– **type**

Returns G_UNKNOWN. This method is meant to be overridden by the subclasses implementing vertices and edges.

3.4. GraphView

INHERITS FROM	View : Responder : Object
REQUIRES HEADER FILES	GraphView.h
DEFINED IN	GraphView

CLASS DESCRIPTION

A GraphView is a subclass of a View providing display, printing and interactive editing capabilities for the basic Graph class. It accepts methods from various components of the interface. To include this class in an application, add it to the project file through the Interface Builder and create a Custom View (of type GraphView) in a window. Other components of the interface can then be set up to send messages to the view as needed.
For each instance of a GraphView there is an underlying graph which is an instance of the Graph class and "governed" by the GraphView.
To speed up the display of the graph, an off-screen window is used for caching. The image can then be composited from this window onto the screen for displaying.
NOTE: Many of the methods described below are sent by user interface objects. Sometimes the GraphView class takes advantage of this and explicitly queries the sender of the method for an argument value. This causes some methods to be dependent on a certain class of sending objects. If this is not mentioned in the description, the sender is ignored and can be any object capable of sending a message.

INSTANCE VARIABLES

Inherited from Object	struct _SHARED	*isa;
Inherited from Responder	id	nextResponder;
Inherited from View	NXRect	frame;
	NXRect	bounds;
	id	superview;
	id	subviews;
	id	window;
	struct__vFlags	vFlags;

Declared in GraphView

```
BOOL      grid;
BOOL      showInfo;
BOOL      edgeLabels;
id        infoField;
id        labelField;
id        typeSwitch;
id        graph;
int       mode;
float     scaleFactor;
id        cacheWindow;
BOOL      cacheing;
```

grid	Is the grid to be displayed or not.
showInfo	Should we display characteristics for graph objects, i.e. vertices and edges.
edgeLabels	Are labels to be displayed for edges as well as vertices.
infoField	Outlet. A ScrollText object for displaying selected objects' characteristics. <<specific for GV>>
labelField	Outlet. A form for displaying labels of selected objects. <<specific for GV>>
typeSwitch	Outlet. A button (checkbox) for indicating whether the graph is directed or not. <<specific for GV>>
graph	The graph itself. See the Graph class for details.
mode	Decides the mode of the GraphView class. See setPointMode: below for details.
scaleFactor	The current scaling factor of the graph.
cacheWindow	An off-screen window for caching.
cacheing	Set to indicate that the graph is to be drawn to the cache and not to the screen.

METHOD TYPES

Creating and freeing a GraphView	+ newViewWithFrame:
	- free
Setting outlets	- setInfoField:
	- setLabelField:
	- setTypeSwitch:
Responding to events	- mouseDown:
	- acceptsFirstResponder:
Drawing	- drawSelf:
	- cacheRect:andDisplay:
	- cacheObject:
Printing	- printGraph:
	- beginSetup
I/O methods	- readGraphFromFile:
	- writeGraphToFile:
	- revertTo:
Target/Action methods	- showInfo:
	- edgeLabels:
	- gridOn:
	- scale:
	- cut:
	- copy:
	- paste:
	- updateInfo:
	- updateLabel:
	- setPointMode:
	- reverse:
	- setType:
	- selectAll:
	- compress:
Cover methods for the graph class	
	- getGraphInfo
	- getDispCharEdges
	- getDispCharVertices
	- setDispCharEdges:
	- setDispCharVertices:
Private methods	- selectObject:shift:
	- unselect

FACTORY METHODS

newViewWithFrame:

+ **newViewWithFrame:**(NXRect *)*aRect*

Creates a new instance of GraphView, with a bounds rectangle defined by *aRect*. The cachewindow is created and the display attributes set to default values: {POINT-mode, no grid, no edgelabels, no characteristics shown, scalefactor=1}.

INSTANCE METHODS

acceptsFirstResponder

– (BOOL) **acceptsFirstResponder**

Returns YES, indicating that the GraphView is always willing to become the FirstResponder.

beginSetup

– **beginSetup**

Overridden method which does the necessary initialization of PostScript variables before printing.

cacheObject:

– **cacheObject:***object*

Draws the object specified in the cachewindow only.

cacheRect:andDisplay:

– **cacheRect:**(NXRect *)*aRect* **andDisplay:**(BOOL)*flag*

Draws all objects which bounding rectangles intersect the rectangle specified by *aRect*. If *flag* is YES the new image is also composited to the screen.

compress:

– **compress:***sender*

This method can be sent by any object and causes the labels of edges and vertices to be compressed. Labels are specified as one or two characters and the following sequence is used. A, B ... Z, a ... z, 0 ... 9, AA, AB ... AZ, Aa ... 99 giving $(26+26+10)^2$ unique labels. Compression maintains the ordering of objects according to labels using the ordering scheme defined above, i.e. if the label of vertex *v* precedes the label of vertex *u* before compression, it will also do so afterwards.

copy:

– **copy:***sender*

This method is normally sent by the copy item of the Edit menu. It copies all selected graph objects to the pasteboard. Note that an edge can only be copied if both adjacent vertices are selected. Also when a vertex is copied, none of its incident edges will be copied unless both the edge and the vertex on the other end of the edge is selected.

See also: **cut:, paste:**

cut:

– **cut:***sender*

This method is usually sent by the Cut item of the Edit menu. It performs the **copy:** method and then deletes all selected objects (whether they were actually copied or not) and objects that might "depend" on the selected objects. Note that cutting a vertex implies deleting all its incident edges also, even though they are not selected, since they depend on the selected vertex.

See also: **copy:, paste:**

drawSelf::

– **drawSelf:**(NXRect *)*aRect*:(int)*nRects*

Overridden method which draws a portion of the graph indicated by *aRect*. If the instance variable *cacheing* is set to YES, this method will draw to the cache otherwise the image will be composited from the cache onto the screen. This mechanism allows us to avoid actually redrawing each part of the graph when scrolling and resizing windows since the graph is actually stored completely in an off-screen cache and needs only to be composited to the screen. Composition is obviously much faster than redrawing, since the hierarchy of the graph doesn't have to be traversed each time.
NOTE: The complete image isn't actually stored in the cache. Edge labels and characteristics shown on the display are drawn after the compositing has been done. This results in saving time when switching the display of these items on and off, but again we lose some time when scrolling, resizing etc. with the display of these attributes turned on.

edgeLabels:

– **edgeLabels:***sender*

This methods queries the *sender* by sending an **intValue** message to it, using the result to decide whether edge labels are to be displayed or not. In the GraphView application this message is sent by a checkbox button. A value of zero means that edge labels are not to be displayed; any other value will cause labels to be displayed.

free

– **free**

Sends a free message to the underlying graph and frees up all other data structures and memory used by the receiver, including the cachewindow.

See also **free** (for the Graph class).

getDispCharEdges

– (char *)**getDispCharEdges**

In the GraphView application the user can dynamically specify which characteristics are to be displayed for the graphs edges and vertices. This is a cover method for the a method with the same name in the Graph class. See the documentation on that class for details.

See also **getDispCharVertices, setDispCharEdges:, setDispCharVertices,** (the same methods in the Graph class).

getDispCharVertices

– (char *)**getDispCharVertices**

See **getDispCharEdges** above.

getGraphInfo

– (char *)**getGraphInfo**

Cover method for the same method of the Graph class. See the documentation for that class for details.

gridOn:

– **gridOn:***sender*

This methods queries the *sender* by sending an **intValue** message to it, using the result to decide whether a grid is to be displayed or not. In the GraphView application this message is sent by a checkbox button. A value of zero means that a grid is not to be displayed; any other value will cause a grid to be displayed.

mouseDown:

– **mouseDown:**(NXEvent *)*theEvent*

Overridden method which handles mouse-generated events. The actions performed depend on the current pointmode. Three modes are possible: when in POINT-mode, the user can point to and select an object by clicking the left mouse button. To unselect an already selected object, the user can press the shift key when clicking. Objects can also be selected in this mode by dragging out a box around them. Furthermore selected objects can be moved around at will by clicking and dragging in the usual way.

ADDVERTEX-mode allows the user to click anywhere (within the frame of the view) and a vertex will be created and assigned an unused label. ADDEDGE-mode allows the user to add edges to the graph by clicking on the source vertex and dragging a "rubber band" to the destination vertex. The edge will be created when the mouse button is released within the bounds of a vertex.

See also **setPointMode:**

paste:

– **paste:***sender*

This method is sent by the paste item in the Edit menu. It copies the contents of the pasteboard into the graph attached to the receiver. The subgraph that is pasted will become the selected part of the underlying graph. It will have the same geometric configuration that it had in the graph it was copied or cut from except coordinates of all vertices copied are translated by a constant amount to avoid coinciding with the destination image if the user is cutting and pasting within the same graph.

printGraph:

– **printGraph:***sender*

This method is sent by a menu item in the GraphView application. It scales the graph so that it will fit onto the currently selected page size and calls **printPSCode:**. If the intelligent print option is on (set through the preferences panel) the method will select landscape or portrait printing according to the geometric layout of the graph.

readGraphFromFile:

– (BOOL)**readGraphFromFile:**(NXStream *)*aStream*

Reads in a graph specification from *aStream,* caches the image and displays the graph. The button used for directed/undirected is set according to the type of the graph read in. If the graph is successfully read in, the method returns YES, otherwise NO is returned.

See also: **setTypeSwitch:, readGraphFromFile:** (the Graph class), **writeGraphToFile:, revertTo:**

reverse:

– **reverse:***sender*

This method is only applicable if the underlying graph is directed. It reverses the direction of all selected edges and re-caches and displays the underlying graph.

revertTo:

– (BOOL)**revertTo:**(NXStream *)*aStream*

Replaces the underlying graph with a graph from the stream represented by *aStream*. The new graph is cached and displayed. If for some reason the graph can't be read in, the old graph is left as it was and NO is returned. If the graph is successfully read in, YES is returned. This method uses **readGraphFromFile:** to perform the read.

See also: **readGraphFromFile:**

scale:

– **scale:***sender*

The *sender* is assumed to be an object responding to a **floatValue** message. In the GraphView application this message is sent by a slider object. The result of the **floatValue** message should be a float in the range of 0.0 to 1.0 representing a common scale factor for both x and y directions (values > 1.0 will also work but give unpleasing results). The scalefactor is stored in the instance variable *scalefactor*. The new image is cached and displayed.

selectAll:

– **selectAll:***sender*

Every object in the underlying graph is selected and the image recached and displayed.

See also: **selectObject:shift:, unselect**

selectObject:shift:

– **selectObject:***object* **shift:**(BOOL)*flag*

Selects or unselects *object*, according to the state of the shift key indicated by *flag*. If *flag* is YES and *object* is already selected, it is unselected. If *object* is not selected, it is made the only selected object if *flag* is NO, but added to the current selection if *flag* is YES. Furthermore, the label and characteristics of the newly selected object are displayed in the fields represented by the instance variables *infoField* and *labelField*.

See Also: **setInfoField:, setLabelField:**

setDispCharEdges:

– **setDispCharEdges:**(char *)*charspec*

See **getDispCharEdges** above.

setDispCharVertices:

– **setDispCharVertices:**(char *)*charspec*

See **getDispCharEdges** above.

setInfoField:

– **setInfoField:***anObject*

Outlet assignment method. Sets the instance variable *infoField* to *anObject*. *Infofield* is a ScrollText object used to display the characteristics of the most recently selected object.

setLabelField:

– **setLabelField:***anObject*

Outlet assignment method. Sets the instance variable *labelField* to *anObject*. *LabelField* is a Form object used to display the label of the most recently selected object.

setPointMode:

– **setPointMode:***sender*

In the GraphView application the sender of this message is the Tools menu. [*sender* **selectedRow**] is used to find out which cell sent the message. Row 0 means ADDVERTEX, row 1 ADDEDGE and row 2 POINT. <<This is bad style and should be replaced by three separate methods.>> The *mode* instance variable is set accordingly.

See also: **mouseDown:**

setType:

– **setType:***sender*

The sender is queried for an **intValue** which is used to determine whether the underlying graph should be displayed as directed or not. In the GraphView application this message is sent by a checkbox button. A value of zero means that the graph is to be displayed as undirected; any other value will cause the graph to be displayed as directed.

setTypeSwitch:

– **setTypeSwitch:***anObject*

Outlet assignment method. Sets the instance variable *typeSwitch* to *anObject*. *TypeSwitch* is a checkbox type Button object used to indicate the mode of the current graph (directed/undirected).

showInfo:

– **showInfo:***sender*

The sender is queried for an **intValue** which is used to determine whether the underlying graph should be displayed with characteristics or not. In the GraphView application this message is sent by a checkbox button. A value of zero means that the graph is not to be displayed with characteristics; any other value will cause characteristics to be displayed.
NOTE: To the particular characteristics that are actually displayed are determined by the values passed to **setDispCharVertices:** and **setDispCharEdges:**.

See also: **setDispCharVertices:, setDispCharEdges:,** and corresponding methods in the Graph class.

unselect

– **unselect**

Private method. Unselects all objects in the underlying graph. Recaches and displays the graph if necessary.

See also: **selectObject:shift:, selectAll:**

updateInfo:

– **updateInfo:***sender*

The receipt of this message causes the GraphView to update the characteristics of the currently selected object (there has to be exactly one object selected) with the contents of the ScrollText represented by the *infoField* instance variable. The object in question is redrawn.

See also: **setInfo:** (the GraphObject class)

updateLabel:

– **updateLabel:***sender*

This message should be sent by a Form object or one that responds to a **stringValue** message. The label of the currently selected object is updated with the result of [*sender* **stringValue**] and the object redrawn. If the new label is not unique in the graph an error panel is displayed.

See also: **updateLabelFor:with:** (in the Graph class)

writeGraphToFile:

– (BOOL)**writeGraphToFile:**(NXStream *)*aStream*

Cover method for corresponding method in the Graph class.

4. File Formats Used by GraphView

Graph files

Graph files are used for storing graphs on disk as well as communicating with the transformation programs. The ".G" suffix used to identify graph files and they also have a special icon, displaying a **G**, in the Workspace Manager's browser.

In the description below, keywords recognized by GraphView are in **bold face** required parameters are written in *<angle brackets>* and optional parameters in *{curly brackets}*. Comments can be inserted in graph files following a semicolon on a separate line. Below, comments are inserted for clarification only.

;;; A graph file should start with three lines describing the type of
;;; the graph, the number of vertices and the number of edges. Each
;;; edge is counted once as an incoming edge to a vertex and once
;;; as an outgoing edge so the entry for the number of edges is
;;; really twice the number of visible edges in the graph.
;;; *<graphtype>* is either d or D for directed or u (U) for undirected
gtype *<graphtype>*
nvertices *<integer>*
nedges *<integer>*

;;; The next entry describes the characteristics associated with
;;; the graph itself.
characteristics
{any text (no comments) }
endtext

;;; The next two entries are lists of keywords separated by white
;;; space, describing the characteristics to be displayed for edges
;;; and vertices, respectively
characteristics-disp-edges
{keywords (no comments) }
endtext

characteristics-disp-vertices
{keywords (no comments) }
endtext

```
;;; The remainder of the file consists of entries for each
;;; vertex in the graph.  Following each vertex entry, are
;;; entries for all edges connected to the vertex.
;;; <vertex#>        unique number in the range 0..nvertices-1.
;;; <#edges>         number of edge entries following.
;;; <label>          one or two characters from the set [A-Za-z0-9].
;;; <x-loc>          floating point number denoting the x coordinate.
;;; <y-loc>          floating point number denoting the y coordinate.
;;; <selected>       1 if the vertex is selected, 0 otherwise.

vertex:<vertex#>:<#edges>:<label>:<x-loc>:<y-loc>:<selected>
{characteristics, any text (no comments) }
endtext

;;; There are two different types of edge entries.  Each visible edge
;;; has an edgeto and an edgefrom entry.  For an incoming
;;; edge there is an edgefrom entry.  These entries contain only
;;; information about connectivity.  Labels, characteristics etc.
;;; are contained in the edgeto entries.
;;; <edge#>          unique number in the range of 0..nedges-1 for
;;;                  this entry.
;;; <edge-to#>       the number of the other part of this edge (an
;;;                  edgeto entry)
;;; <vertex#>        The number of the source vertex.

edgefrom:<edge#>:<edge-to#>:<vertex#>


;;; edgeto entries are contain the actual information associated
;;; with the edge
;;; <edge#>          unique number in the range of 0..nedges-1 for
;;;                  this entry.
;;; <edge-from#>     the number of the other part of this edge (an
;;;                  edgeto entry)
;;; <vertex#>        The number of the destination vertex.
;;; <label>          one or two characters from the set [A-Za-z0-9].
;;; <selected>       1 if the edge is selected 0 otherwise.

edgeto:<edge#>:<edge-from#>:<vertex#>:<label>:<selected>
{characteristics, any text (no comments) }
endtext
```

NOTE: GraphView guarantees that the edges adjacent to a vertex will be written in order, reflecting the embedding of the edges around the vertex. This applies also to transformation programs, that write their output files using the I/O methods for the basic graph class. Currently this order is clockwise around the vertex, starting from the "three o'clock" position.

History files

History files are used to store the creation history of each graph. They are created and maintained automatically by the GraphView application. History files have the same name as the corresponding graph file, but a ".H" suffix instead of ".G". Each file consists of a sequence of lines of the form:

<parent graph> <name of transformation> <optional parameters>

An example would be:

/u/me/Graphs/digraph.G dfs.X -v A

A history file is not created until a transformation is applied for the first time.

Version files

Version files are used to store the current version number for each graph. They are created and maintained automatically by the GraphView application. When a transformation is applied to a graph digraph.G, the resulting graph will automatically be named digraph_1.G. Applying a transformation to this graph will yield digraph_2.G, and so on. The last sequence number used is kept in a version file with the same name as the parent graph and a ".version" suffix. In our example the version file would be named digraph.version. Version files contain a single line of the form:

version <last version number>

A version file is not created until a transformation is applied for the first time.

Transformation files

Transformation files are executable programs, launched by the GraphView application. GraphView assumes that transformation programs have a ".X" suffix and they are displayed with an **X** icon in the Workspace browser.