# GraphTool - A Tool for Interactive Design and Manipulation of Graphs and Graph Algorithms

Written by Vitus J. Leung    Michael B. Dillencourt    Andrew L. Bliss

Portable Version 2.0
March 31, 1997
University of California, Irvine

## 1 Overview

**GraphTool** is an interactive tool for editing graphs and visualizing the execution and results of graph algorithms. It runs under the X Windows environment and has a full window/mouse interface. In addition, there is a standalone program called the **Wrapper** which simulates the **GraphTool** interface without graphics for batch processing of graph algorithms.

While the primary purpose of **GraphTool** is to provide a means for experimentally investigating the performance of graph algorithms, it has other useful features as well.

It provides features for printing graphs in a visually appealing format, which makes it easier to prepare papers for publication. It also provides a facility for "animating" algorithms, which means that it can be used in computer assisted instruction (CAI) and for preparing video presentations of algorithms.

## 2 GraphTool's Capabilities: A Quick Summary

### 2.1 Graph Definition and Modification

**GraphTool** runs in several modes, during which the left mouse button takes on different meanings to enable you to define and modify a graph using just the left mouse button.

MOVE — Clicking on a spot outside an existing node causes a new node to be created and placed there. If the mouse button is held down with the pointer over a node, the node is picked up and can be moved by moving the mouse button. All edges connected to the node are redrawn as the node is moved.

If the shift key is held down when creating a node, **GraphTool** will allow you to enter exact coordinates for the node.

CONNECT — Edges are defined by clicking on the first node and then the second node. If the graph is directed, the node is drawn with an arrow at the second node to show that it goes from the first node to the second node.

DELETE — Delete removes both edges and nodes. Just click on a node or near an edge and it is deleted. For nodes with edges, the node won't be deleted unless you force **GraphTool** to do so by holding down the shift key.

Every node and edge has a floating-point weight that is set from a global weight when a node or edge is created. The weight for a node or edge can be modified at any time. The global weight may also be modified at any time so that new objects will have different weights from those already created. The global weight defaults to 1.

Nodes and edges also have visual attributes: colors (patterns in monochrome) for nodes and widths and colors (dotting/dashing in monochrome) for edges. These attributes are taken from global values just as for weights and may be changed in the same way. There are eleven different node interiors and six different line patterns in monochrome and lines may be any number of pixels thick.

Finally, nodes and edges are numbered for labeling purposes. You don't have direct control over what number each object has but you can affect the entire numbering system. **GraphTool** automatically assigns a number to each new node or edge as it is loaded from a file or added in the canvas. The graph can be renumbered on demand and is renumbered as a side effect of several commands.

## 2.2 Graph Display and Printing

The current graph is stored with full floating point resolution so that it may be rescaled and panned at will. The apparent size of a graph may be doubled or halved, allowing you to view an entire graph or just a small portion. The view may be moved around to an arbitrary point in the graph when the graph is larger than the screen can show. The only limit on the resolution that may be displayed is the machine accuracy.

Printing is provided through a facility that converts a description of a graph into a standard Unix format (PIC). The graph may then either be printed directly, using standard Unix software, or incorporated into documents using standard Unix conversion routines that convert it into Troff or TeX format.

## 2.3 Application Programs

The most novel aspect of **GraphTool** is the capability it provides for integrating application programs. (In this context, application programs are programs that create, manipulate, or compute characteristics of graphs.) Application programs are selected by a menu. The menu is driven by a text file, so as new programs become available, incorporating them into the menu is straightforward.

Once an application program is invoked by **GraphTool**, the application program communicates with **GraphTool** by sending it commands. The commands may be graph modification commands (e.g., add a new edge, delete a vertex, change an attribute), or they may be selection commands. Selection commands cause **GraphTool** to ask the user to select a vertex or edge; the user's selection is then passed back to the application program.

As an illustration of the interplay between an application program and **GraphTool**, consider an application that computes the shortest path between two selected nodes in a graph. First, the application would ask **GraphTool** to ask the user to select two vertices. Then the application would compute the shortest path between the two selected vertices and cause **GraphTool** to display the path (e.g., by highlighting all the edges on the path).

Now suppose that we would like to visualize how the application works, rather than just observe its output. This could be done by changing the visual attributes of each edge that is placed on a candidate path, perhaps by making it dotted rather than solid, and changing it back if the edge is removed from consideration. Moreover, one of **GraphTool**'s commands allows you to specify a delay for it to wait between later commands so that you can easily control the display speed of the algorithm. This is useful for debugging, and also for educational and presentation purposes.

In addition to **GraphTool**'s visual interface, the **Wrapper** provides the same graph management capabilities in a command-line driven package to facilitate batch mode processing of algorithms. The **Wrapper** is designed to be easily run from shell scripts so complicated or repeated executions can be entirely automated. For example, if you have an algorithm which computes the convex hull of a set of points and you want to test it on a large number of random point sets, this would be very tedious in **GraphTool**. However, using the **Wrapper** this entire task can be automated. The algorithms to generate random graphs and compute the convex hull can be developed and debugged in **GraphTool**, and then, without any code changes, can be linked together in a shell script with the **Wrapper** for any number of test runs.

# 3 A Detailed Description of GraphTool

## 3.1 A Note on Windowing Systems

This generic description should not cause any difficulties unless your local setup is radically different from the setup under which **GraphTool** was developed. In particular, X Windows systems vary widely in window appearance, colors, cursors and sizes, but the basic interface should still be preserved.

## 3.2 Working Areas

When **GraphTool** is first executed, it creates two windows. The large, blank window is the *canvas*. The canvas is the work area in **GraphTool**, where the graph will be displayed and edited. The canvas has *scroll bars* on it that let you move around the graph if the graph is too large to fit in the canvas window all at once. A scroll bar looks like a rectangular area, the bar itself, with another rectangular area inside the bar, called the *thumb*. The size of the thumb indicates the size of what you see in the canvas as compared to the size of the graph, and the position of the thumb indicates where you are in the graph. A good way to think about it is that the canvas is a window on the entire graph, and the scroll bars indicate the size and position of the window in the graph. When **GraphTool** is started, the graph is the same size as the canvas, so the thumb is the same size as the scroll bar. When you double and halve the graph, the size of the thumb will change to show you the new size of the window that the canvas represents. To move your viewpoint, simply move the mouse into the scrollbar and select the thumb. Then drag the thumb to the position where you want it and let up the mouse button. The new view will then be drawn.

The smaller window that contains all of the controls for **GraphTool** is the *control panel*. The canvas and control windows are completely independent of each other, and can be moved, resized or closed at whim.

It is very important to remember that the windows are independent when typing. If your system is set up so that keyboard input goes to the window underneath the cursor, you may inadvertently type something into the canvas window that you meant to type into the control panel and vice versa. Make sure that you move the mouse pointer into the appropriate window before you type something. If you type something and you don't get the result you expected, see where the mouse pointer is.

Another important thing to remember is to finish actions that you started in the canvas window. If you are moving a vertex and you drag the pointer outside of the window and let up on the mouse button, when you move back into the canvas **GraphTool** will be confused. As long as you keep the mouse button down, you can move back into the canvas window and resume your activity.

## 3.3 Temporary Objects

Occasionally **GraphTool** will create a temporary object for you to enter some special information in or make a special selection from. There are two types: popup menus and popup windows.

The only popup menu provided by **GraphTool** is the canvas menu, activated by holding down the right mouse button in the canvas. The canvas menu allows you to select an action (section 3.5) or to run an algorithm (section 4) by moving the mouse over one of the menu's items.

**GraphTool** has several popup windows. Popup windows are small windows that appear on top of everything in the center of the screen when **GraphTool** needs a specific piece of information. Nothing else can be done in **GraphTool** while a popup window is active. There are popup windows for printing, entering coordinates for a vertex and for algorithm input. These windows are discussed in more detail in the sections for printing, 3.6, moving, 3.5, and algorithm I/O, 4.4.

## 3.4 Working with GraphTool's Vertices and Edges

### 3.4.1 Appearance

The sample graph shown in Figure 1 illustrates, as closely as can be printed, what a directed graph looks like in **GraphTool**'s canvas. **GraphTool** represents vertices as circles which have an interior style such as solid, hollow, empty, or a color (pattern in monochrome). Edges are represented as lines with a thickness and a color (style such as solid, dashed or dotted, in monochrome). Note in the sample graph how some of the nodes are patterned and the edges vary in both width and style. The appearance of nodes and edges is controlled by fields on the control panel as described in section 3.6.

### 3.4.2 Selection

To work with **GraphTool**'s objects you need to know how **GraphTool** thinks of them and their relationship to the mouse pointer. The mouse pointer shape has a selection point, called a *hot spot*, which determines the coordinates of the mouse pointer given to the program. As this hot spot can change from pointer to pointer, it is important to know where it is so that you select the things you want to select. This information is given in section 3.5 where **GraphTool**'s various pointer shapes are described.

In addition to the selection point, you also need to know how **GraphTool** looks for the object that you indicate when you click on it. **GraphTool**'s notion of object area for a vertex is the entire circle, as you would expect. For edges, there is a "gravity field" around the edge of a few pixels. If the pointer is in the gravity field of an edge, that edge is selected, so it is not necessary for the pointer to be precisely on an edge to select it.

When checking its lists of objects for the area hit, **GraphTool** searches its entire list of vertices first, and then checks the edges, so don't select an edge while you are partially in a vertex. If two object's areas overlap, **GraphTool** will pick the one that it encounters first. To the user, having no knowledge of the order in which **GraphTool** is storing objects, this will seem arbitrary. In short, use unambiguous portion of objects for selection.

When an object is successfully selected, depending on the mode of operation, its weight and appearance factors will be set in the control panel. The last selected object is remembered

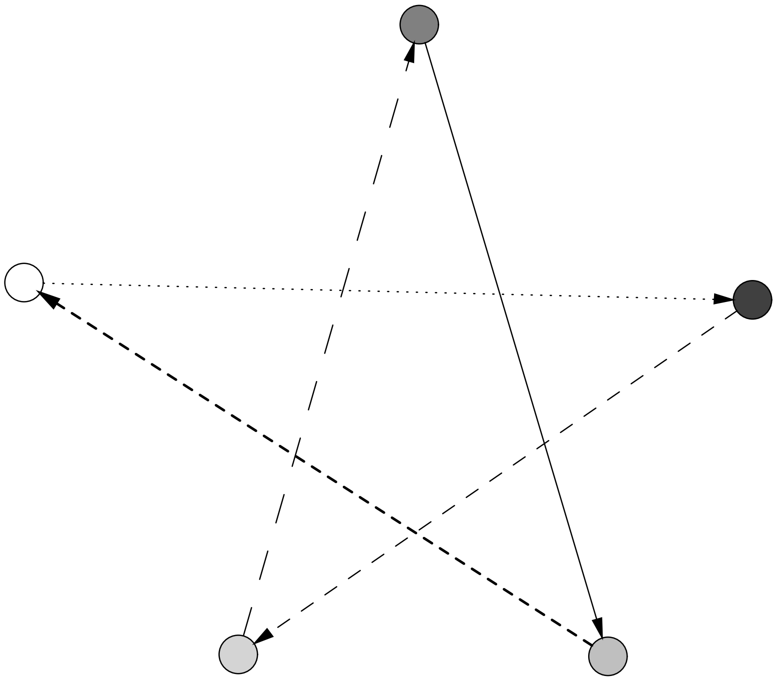

Figure 1: A Sample Graph

for some operations. Section 3.5 describes precisely when this information is set. Various commands will refer to the *current object*; this is the last selected object.

#### 3.4.3 Numbering

**GraphTool** maintains a number on each node and edge in the graph for naming purposes. **GraphTool** automatically assigns these numbers as nodes and edges are created, so you can't directly type them in like a weight, but you can control the overall numbering scheme.

Numbering begins when a node or edge is created, either by you creating a new one in the canvas or because you are loading a file. The first node and the first edge are both assigned the number one and the numbers increase from there. If you delete a node or edge, its number is not re-used, so holes in the numbering system can occur. If your numbering becomes too fragmented, you can force **GraphTool** to renumber the nodes and edges from one again with the Number button on the control panel (section 3.6).

Note that **GraphTool** itself does not use these numbers; they are only for you to use to reference things. Also note that while **GraphTool** saves the weights and appearances of nodes and edges, it does not save their numbers, so graphs are always numbered from one after loading. Finally, under certain circumstances **GraphTool** will renumber the graph automatically as a side effect of some action (e.g., saving a graph).

There are ways you can create graphs with specific numberings but they involve separate coding. See section 4 (section 4.4 in particular) for information on how to write a program to set up and manipulate a graph with custom numbers.

### 3.5 Doing Things

**GraphTool**'s interface is largely controlled by the left mouse button. There is a *current action* associated with the left mouse button that tells **GraphTool** what your latest mouse click will mean. The name of the current action is displayed in the Action field in the control panel and is also indicated by the mouse pointer shape. Actions fall into three rough categories: for editing graphs, when running algorithms and for selecting a rectangular area.

**GraphTool**'s editing actions are used for creating and modifying graphs. They give you control over creation, deletion and placement of nodes and edges in the graph. You may also get and set appearance and weighting information.

CONNECT

CONNECT starts a connection between two vertices. Once a vertex is selected, the action is automatically changed to COMPLETE. If you accidentally select a node while in CONNECT mode and you don't want to complete the connection, undo will cancel the completion and put you back in CONNECT.

Selecting a node in COMPLETE finishes a connection between two vertices. Self-edges are not allowed, but duplicate edges are. However, duplicate edges are not marked in any way so you can not tell how many edges are between two nodes just by looking at them. The current action is automatically changed to CONNECT after COMPLETE.

The mouse pointer for CONNECT is a circle with an arrow coming out of it. The selection point is the center of the circle. CONNECT can be selected from the canvas menu or by pressing 'c' in the canvas window.

The mouse pointer for COMPLETE is a circle with an arrow going into it. The selection point is the center of the circle.

If a part of the canvas outside of any existing vertex is selected, a vertex will be created.

DELETE

DELETE removes either vertices or edges, depending on which kind of object is selected.

When deleting vertices, **GraphTool** will not allow a vertex to be deleted unless it has no edges connected to it. If you wish to over-ride this safety feature, hold down the shift key when selecting the vertex. The vertex and all edges connected to it will be deleted.

The mouse pointer for DELETE is an axe, with the selection point in the middle of the blade. DELETE can be selected from the canvas menu or by pressing 'd' in the canvas window.

INFO — INFO displays the number, weight and coordinates of vertices and the number, weight and numbers of the connected nodes for edges. The weight and appearance attributes are set in the control panel.

The mouse pointer for INFO is a question mark. The selection point is the center of the circular portion of the question mark. INFO can be selected from the canvas menu or by pressing 'i' in the canvas window.

MOVE — MOVE allows you to drag vertices to new locations. Move the mouse pointer inside the vertex you want to move and press and hold the left mouse button. If the vertex is not hollow or empty, it will be redrawn hollow. All edges that are connected to the vertex and are not solid, one-pixel-wide lines will be redrawn as solid, one-pixel-wide lines. You have now picked up the vertex and you can move it by moving the mouse around without releasing the left mouse button. The coordinate display in the control panel is updated as the mouse moves so that you know where you are in the graph's coordinate space.

If you are holding down the shift key when you release the left mouse button, a small window will pop up in the center of the screen with the current coordinates of the vertex. You can edit these coordinates with the keyboard if you wish to enter a precise location. When you are finished, select the Done button to move the vertex to the input location or select Cancel to leave it where it is. It is not necessary to hold the shift key after the window has popped up.

The mouse pointer for MOVE is a left-pointing wedge, with the selection point at the tip of the wedge. MOVE can be selected from the canvas menu or by pressing 'm' in the canvas window.

If a part of the canvas outside of any existing vertex is selected, a vertex will be created.

SET — SET without shift selects a node or edge. In this mode the weight and appearance attributes are set in the control panel.

SET with shift applies the current weight and appearance to the selected node.

The mouse pointer for SET is a delta, or triangle. The selection point is the top point of the delta. SET can be selected from the canvas menu or by pressing 's' in the canvas window.

When running an algorithm (see section 4), **GraphTool**'s editing actions are unavailable and a new set of actions takes their place. These actions indicate that an algorithm is running, allow an algorithm to get node and edge selections from the user and let an algorithm wait on a mouse click from a user.

EDGE EDGE is entered when an application requests the user to select an edge. The current action is set to RUN after an edge has been selected.

The mouse pointer for EDGE is an arrow. The selection point is the tip of the arrow.

NODE NODE is entered when an application requests the user to select a node. The current action is set to RUN after a node has been selected.

The mouse pointer for NODE is an arrow. The selection point is the tip of the arrow.

RUN RUN indicates that an application is being run and most of the normal editing commands are disabled.

The mouse pointer for RUN is a rectangle with some lines in it, intended to indicate some kind of code. The selection point doesn't matter because nothing is selected in RUN.

WAIT WAIT is entered when an application is waiting for the user to click a mouse button. RUN is re-entered after WAIT.

The mouse pointer for WAIT is a picture of the mouse. The selection point doesn't matter as nothing is selected in WAIT.

The last of **GraphTool**'s actions is for selecting a rectangular area:

BOXES BOX UL is entered when **GraphTool** wants you to select a rectangular region of the screen, such as when appending a graph to the current graph. Move to the point on the screen where the upper-left-hand corner of the area should be and press down the left mouse button. The action will change to BOX LR.

In BOX LR, you select the lower-right corner of the box to complement the upper-left corner you just selected. You can drag the mouse around with the left button down and pull out an outline of the region. When the region is where you want it, let up on the mouse button and **GraphTool** will continue processing with the area you selected.

The mouse pointer for BOX UL is a box with the upper-left-hand corner highlighted. The selection point is the center of the box.

The mouse pointer for BOX LR is a box with the lower-right-hand corner highlighted. The selection point is the center of the box.

## 3.6 The Control Panel

The control panel is the area in which most of the information about the current graph is displayed and entered. It also contains some control settings and buttons for activating certain functions. Note that all input described here must take place within the control panel. In the following description, the type of input device which a piece is made of is given after its name. The devices are:

A text field is an area in which text can be entered. The field is selected by clicking in its entry area. When a field is selected, a caret is placed at the current entry point. More text can be typed in a text field than may fit on the screen; if there is more text the text scrolls left and a left-pointing arrow is displayed by the title. It is not necessary to hit return in these fields; in fact, return may initiate a special action.

A message field displays a line of text.

A check box represents a boolean action and looks like a small rectangle with or without a checkmark in it. Check boxes can be set and reset by clicking somewhere on it.

A button activates something when clicked on and looks like a rounded rectangle with a name in it.

A cycle is a list of items that can be cycled through for a multiple choice item. Note that this use of the word "cycle" has nothing to do with the concept of a cycle in a graph; it is used because it is what X Windows calls this particular type of input. A cycle has a label followed by a picture of two arrows cycling around followed by the current item. Clicking somewhere on it causes the value to change. The left button goes forwards and the right buttons goes backwards.

**Action** **(Message)** This displays the current action. The mouse pointer also changes to indicate the current action.

**Append** **(Button)** Append loads the graph specified in the Filename item from its file and merges it with the current graph. You select the area in which you want the graph to go and **GraphTool** scales and offsets the nodes appropriately. All header information in the appended graph is ignored so the direction, zoom level, etc. of the original graph is unchanged.

When this button is clicked, BOX UL mode is activated. You can then pull out the box that you want the new graph to go in as described under the BOX UL and BOX LR modes. When you have selected the box, **GraphTool** loads in the new graph and fits it into the given box. The scaling may change the aspect ratio of the new graph from its original value.

The graph is renumbered from one as a side-effect of appending.

**Clear** **(Button)** Clear removes the current graph and resets many of **GraphTool**'s parameters such as the coordinate system and zoom level. If the current graph has unsaved modifications you will be alerted of it and given the option to go ahead anyway or to cancel the clear.

Clear empties the undo buffer (See Undo).

**Coords** **(Message)** The coordinates of the mouse pointer in the current graph's coordinate system.

**Directed** **(Check Box)** This on-off item controls whether the current graph has directed edges or not. **GraphTool** does not differentiate between directed and undirected edges; this item only controls their appearance. Therefore, if you toggle this item from directed to undirected and then back, the graph will be just as it was before. Information is neither lost nor gained.

**Double** **(Button)** Double doubles a scaling factor in the coordinate system so that the screen views only one fourth of the area that it did previously. The quarter of the screen at the center becomes the new view. The scroll bars are reset to indicate the new size of the screen and the new position of the view. The node radius does not increase with the doubling level, so the size of the nodes on screen does not indicate the current zoom level.

Note that actual coordinates of objects do not change; only a scaling factor in the view transformation. The appearance of the graph changes but not the actual coordinates.

The view can only be doubled a certain number of times because of machine limits. When the screen can no longer be doubled and and a doubling attempt is made, an error message is displayed in the Message field.

**Edges** **(Cycle)** In monochrome, Edges is a graphic field that shows what the current line style will look like on a miniature line segment next to the name. All new edges are created with the line style in this field.

If the current action is SET and this field is cycled, the line style of the current edge will be changed to the new value in this field.

In color, Edges shows an abbreviation for the current line color or color combination on a miniture line segment next to the name.

**EW One** **(Button)** EW One stands for Edge Weight One. The weights of all edges in the graph are set to one.

**Filename** **(Text)** The current filename. A filename must be present before a graph can be loaded or saved.

**Fit** **(Button)** Fit draws a box around the current graph and manipulates the scaling factors in the coordinate system so that the graph just fills the screen. The zoom level is reset to a base level so that halving and doubling have the same effect as starting from scratch. The scroll bars are reset to reflect the fact that the entire view is now visible.

The entire graph is guaranteed to be visible, but the scaling is isometric so one axis may fill the screen while the other axis takes up less than the entire screen in its direction.

There is a small margin left around the screen so that vertices do not lie precisely on the edge.

**Halve** **(Button)** Halve does the opposite of Double. A coordinate system scaling value is halved so that the screen views four times as much area as before. The view shrinks around the center of the screen. As with Double, the actual coordinates of objects do not change; just a scaling factor.

The view can be halved indefinitely, but if you halve too much, you may not be able to double enough to enlarge your graph to a reasonable size. Use Fit to reset the zoom level.

**Load** **(Button)** Loads the current file specified in the Filename field. If the current filename does not have an extension a '.g' extension is automatically appended. If there is no current filename, **GraphTool** will tell you to type one in before loading. If there is a system error when loading the graph file, **GraphTool** will display the standard system error message for it in the Message field. If the file is not what **GraphTool** expects, **GraphTool** will display the message `File format error. See error file` in the Message field. The error file is a text file which will contain some messages from **GraphTool** describing what it doesn't like about the file. The error file name is made from the current file name with .err appended. For more information on **GraphTool**'s file format, see section C.

If a current graph exists and has unsaved modifications, an alert will pop up asking you whether you really want to load the file or not. If you choose to load the file, you will destroy the current graph. If you cancel the action, the load will be aborted and the current graph will be unchanged.

Loading a file empties the undo buffer (See Undo).

**Messages** **(Message)** The display area for all messages. Messages are blanked after a certain number of canvas events like mouse movement, button presses, etc.

**Nodes** **(Cycle)** In monochrome, Nodes is a graphic field which displays what the current node fill style looks like in a small picture of a node by the title. Note that there are two styles, hollow and empty, which look very similar in a graph. Hollow is indicated by an empty circle. A hollow node in the graph is drawn without a center so it may overlap other nodes without erasing them. Empty nodes are indicated by a hollow node with an X in the center, indicating that the center is filled with the background color, erasing whatever was there.

All new nodes are created with this fill style. If the current action is SET and this field is cycled, the fill style of the current node is changed to the value of this field.

In color, Nodes shows an abbreviation for the current node color or color combination in a small picture of a node by the title.

**Number** **(Button)** Renumbers the nodes and edges in the graph starting at one and increasing.

**NW One** **(Button)** NW One stands for Node Weight One. The weights of all nodes in the graph are set to one.

**Plain** **(Button)** Removes all widths and colors (patterns and line styles in monochrome) from the current graph. They are permanently lost.

**Print** **(Button)** Print formats the current graph and writes a file in PIC format that may be printed directly or incorporated into another document.

When Print is clicked on, a popup window comes up with the options for printing. The 'To file:' field holds the name of the file that the print commands will be put in. It defaults to the current file name with an extension of .pic. If there is no current file name, print.pic is used.

There are several check boxes after the file name which control which elements of the graph, such as number, weights and styles, are printed. Node numbers and weights are placed in the center of the node without regard to the size of the node so they may not fit. Node patterns are printed as grey scales and may not look like what you see on the screen. If node patterns are disabled, the nodes are hollow.

Edge numbers and weights are placed at the midpoint of the edge and may overlap it. If the edge is thick, it may obscure the numbers entirely. Edge widths and styles are only approximated, so the appearance may be quite different from the screen representation. If edges styles are disabled, the lines are solid.

The node radius field controls the size of the nodes drawn. The width and height are the width and height of the box that the current graph will be scaled to for printing. The graph itself, not the canvas view, is what is scaled, so if your graph does not fill the canvas, you won't get a lot of blank space around the edges; only the graph is considered. Conversely, if your graph is huge, you won't see just a part of it. It will be scaled to fit in the size you specify.

Once you are satisfied with the print settings, click the Print button in the popup and the print file will be created. Your settings, except for the filename, will be saved for the next time you print. If you cancel, the settings are not changed from what they were when the print popup first came up.

As mentioned previously, the file is in PIC format, and is suitable for conversion to TEX format via tpic (or your favorite utility). It can also be used in the normal fashion as a troff preprocessor input file. If you want to edit your graph for some reason, such as moving things or adding annotations, there are programs that provide editing capabilities for PIC files. You may have to convert the PIC format to a different format that the editor uses, but PIC is a well-known format and there are many translation and editing tools for it. Unfortunately, none of these programs are standard UNIX commands, so their availability will be site-dependent.

A side effect of printing is that the current graph is renumbered from one.

**Quit** **(Button)** Quit is a "safe" exit. If the current graph has unsaved modifications you are prompted as to whether you really want to quit or not. If you quit anyway the current graph will be lost.

**Radius** **(Text)** The current node radius in pixels. When return is pressed in this field, the graph is redrawn with the node radius set to the value in the field.

This value cannot be less than one.

**Redo** **(Button)** Redo reverses what Undo just reversed. In other words, it restores the action that you just undid. See Undo for a complete description of how **GraphTool** manages Undo and Redo.

**Save** **(Button)** Saves the current graph in the file specified by the Filename field. If the current filename does not have an extension a '.g' extension is automatically appended. Save handles no filename and system errors in the same way as Load. There is no way to get format errors when saving. If the specified file exists the user will be alerted of it. You are given the option of saving over, and thus destroying, the previous file or cancelling the save.

Nodes and edges are renumbered consecutively from one as a side effect of saving.

**Undo** **(Button)** Undo reverses the last action performed. Actions that undo reverses are: addition and deletion of nodes and edges, moves of nodes and sets of nodes and edges.

When undoing a node or edge deletion, the number will not be preserved because the graph may have been renumbered in the interim. The recovered node or edge will have the greatest number in the graph.

Undo and Redo use a thing called the *undo buffer*, which is a storage place of the last few actions you did. When you create a new node or edge, move a node, delete an edge or any of the above actions, it is stored in the undo buffer. When you undo, the current action (which, if you just started undoing, will be the last action you performed) in the buffer is reversed and the buffer pointer is stepped down one. Multiple undo's will keep reversing actions and stepping earlier and earlier in the buffer until there is nothing left to undo, at which point **GraphTool** will complain. When you Redo, the current action in the buffer is restored and the buffer pointer is stepped up. You can step all the way back to where you started, if you like, and everything will be back the way it was before you began undoing. The upshot of all this is that your last few actions are stored and you can walk back and forth through them at will.

If the buffer is full, the oldest action is discarded is when a new action is put in the buffer. Also, if you undo down into the buffer so that there are some actions above the current point in the buffer, and then do something new, all events in the buffer above your current position are discarded.

Note that because the buffer contains multiple actions, you cannot "undo an undo." You can only exhaust the buffer of things to undo. Use redo to "undo an undo."

Currently, the undo buffer is twenty-five actions deep. It is cleared when you load or clear or run an algorithm.

In addition to the major function of moving through the undo buffer, undo will also cancel a half-complete edge connection. This will put you back in CONNECT mode. This action does not affect the undo buffer, so you can't redo to get back into COMPLETE.

**Weight** **(Button)** Weight sets the weights of the edges in the graph to the Euclidean distance between the centers of the nodes that the edge connects.

**Weight** **(Text)** The current weight for nodes and edges. This item can be modified at any time as long as the return key is not pressed. If the return key is pressed and the current action is SET , the weight of the current object is set to the value in this field.

The weight of an object is a floating point value, so decimal values may be entered here.

**Width** **(Text)** Width controls the width of new edges and can be used to change the width of existing edges. All new edges are created with a width of this many pixels, which cannot be less than one.

If the current mode is SET and return is pressed in this field, the current edge will have its width set to this value.

### 3.7 The Keyboard

A few notes on keyboard commands: case is significant and all keyboard commands must be typed in the canvas window. The convention 'Ctrl-key' means that the Control key must be held down when pressing the key.

**Ctrl-c** If there is an algorithm running, sends SIGINT (the same signal that is sent when Ctrl-c is detected by a program running normally) to the currently running algorithm. For programs with standard signal handlers (i.e. almost all), this causes the process to terminate. If you have an algorithm that is misbehaving or is not programmed to terminate, this command will kill it.

**Ctrl-l** Performs the same action as the Load button.

**Ctrl-p** Toggles the control panel on and off. If you don't need the control panel or it is getting in the way, you can turn it off. You can then bring it back with another Ctrl-p.

**Ctrl-q** Performs the same action as the Quit button.

**Ctrl-r** Causes the current graph to be redrawn. This is an easy way to remove any garbage left over from a faulty run of an algorithm.

**Ctrl-s** Performs the same action as the Save button.

**Ctrl-v** Prints the current version string in the Message field.

**Ctrl-x** Exits **GraphTool** immediately. There may or may not be confirmations, depending on what your windowing system does. Do not depend on there being any.

**c** Selects CONNECT.

**d** Selects DELETE.

**i** Selects INFO.

**m** Selects MOVE.

**r** Performs the same action as the Redo button.

**s** Selects SET.

**u** Performs the same action as the Undo button.

## 4 Algorithms

External programs that adhere to the following input/output format can be run under **GraphTool** and manipulate the current graph. **GraphTool** gives the current graph to the program in the file format described in section C and then waits for commands from the program. The program may modify the current graph by adding and deleting objects, changing appearances and even drawing its own graphics. The programs are referred to as *algorithms* and also *children*. In addition to any native programming that you may wish to do, there is a library that handles many common functions used in algorithms. It is described in section 4.5.

Algorithms are made visible to **GraphTool** by adding an entry for them in the .graphrc file. **GraphTool** reads the .graphrc file every time the canvas menu is brought up, so the .graphrc file can be modified without having to exit **GraphTool** and re-enter. The entries in the .graphrc file are added to the canvas menu and can be selected from it just like any other menu item.

## 4.1 The .graphrc File

Like graph description files, the .graphrc is line formatted. The types of lines in a .graphrc are:

`#submenu <string menu name>`

Declares a submenu that will be a pull-right from the parent menu. The submenu will be shown with the given name.

`#endsub`

Ends a submenu so that later entries are added to the parent menu. Every #endsub must have a #submenu before it.

`An algorithm entry`

An algorithm entry has no preceding keyword and consists of these two lines:

- The name of the algorithm to show on the menu.
- The relative path of the executable from the current directory and any arguments that you want to pass the algorithm on the command line.

The name of the algorithm will be placed in the current menu for selection. The relative path is kept by **GraphTool** so that it knows what to run when the algorithm's entry is selected in the menu. The path may not begin with a '#' or a '*'. Any space-separated chunks of text after the path are assumed to be command line arguments and will be given to the algorithm with the normal argc-argv convention when the algorithm is executed. Because the space character is the delimiter, you cannot have a space in the path itself and you cannot pass arguments that are a single argument but contain spaces.

When an algorithm entry is selected from the menu, the algorithm is run under **GraphTool** and may interact with **GraphTool** in a variety of ways. A complete description follows.

`A command entry`

A command entry has no preceding keyword and consists of these two lines:

- The name of the command to show on the menu.
- The command to be interpreted.

The name of the command will be placed in the current menu for selection. The command is a GraphTool language command as described in section 4.4 and therefore must begin with a '#'. The leading '#' is the only way that **GraphTool** can determine that the entry is a command entry so be very careful to include it.

When a command entry is selected from the menu, the language command is executed precisely as if an algorithm had given the command to **GraphTool**. This enables you to execute certain actions from the menu for your convenience. For example, there is a language command to remove all edges from the current graph. If you want this function on your menu for easy selection, put a command entry for the function in your .graphrc, as shown in section 4.2.

`A file entry`

A file entry has no preceding keyword and consists of these two lines:

- The name of the file to show on the menu.
- The path of the file to be loaded, prefixed by a '∗'.

The file name given will be placed in the current menu for selection. The file path is a UNIX path, relative to the current directory. The preceding '∗' is stripped from the filename when used; it is only there to identify the path as a file path. The '∗' is the only way that **GraphTool** can determine that the entry is a file entry so be careful to include it.

When a file entry is selected from the menu, **GraphTool** puts the given filename in the Filename text item and attempts to load the file, just as if you had typed the file name in yourself and clicked on the Load button.

Whitespace at the front of the line is ignored so it is possible to indent submenus to show the menu hierarchy clearly.

There is no set limit on the number of entries that may be added to the canvas menu; it is a function of the window system's constraints. Submenus have no set nesting limit; again, this is controlled by the window system. Finally, lines cannot be longer than 255 characters.

## 4.2 An Example .graphrc

```
Remove Edges
#noedges
#submenu Directed
    Shortest Path
     algs/spath -directed
    Any Path
     anypath
    #submenu One
        One Algorithm
         algs/one
    #endsub
#endsub
#submenu Undirected
```

```
    Shortest Path
     algs/spath -undirected
#endsub
#submenu Graph Files
    Convex Hull
     *chull
    Shortest Path
     *../gfiles/spath
#endsub
```

## 4.3 Running Algorithms

When running an algorithm, there is a program which maintains the current graph and responds to commands that is called the *server*. This is either **GraphTool** or the **Wrapper**. **GraphTool** provides the fully interactive environment explained previously, while the **Wrapper** is intended more for a batch environment and does not provide any of the editing and manipulation features of **GraphTool**. The **Wrapper**'s only function is to provide support for the complete algorithm command set and respond in as similar a manner to **GraphTool** as possible while also being able to run completely unattended.

The algorithm is called a *client* because it depends on functionality that the server provides to it. The client may keep its own copy of the current graph for use, but it is wholly dependent on the server for both display of the current graph, if the server does so, and all input that the client may want. On the other hand, the client may be run under any server without any changes, and yet still take advantage of the unique features that server offers.

Therefore, it is important to be familiar with exactly what each server does and which one you should choose to run your algorithm.

### 4.3.1 GraphTool as Server

When an algorithm is invoked, the undo buffer is discarded and the graph is marked as not being saved. Even if the algorithm does nothing, the graph is still marked as having modifications. All editing actions are suspended and RUN mode is entered. If **GraphTool** cannot execute the algorithm for some reason, error messages will be displayed in the Messages field.

When the algorithm exits on its own, the message `Child process exited with code 0` will be displayed in the Messages field to let you know that the algorithm is no longer running. The code number is the exit value of the program, and will not necessarily be zero. If the algorithm was killed by a signal, like a segmentation violation or you killing it with Ctrl-c, **GraphTool** will display a message of the form `Child killed by SIGSEGV` where the actual signal name will be displayed to inform you why it died.

The action in effect before the algorithm was run will be reinstated.

### 4.3.2 The Wrapper as Server

The **Wrapper** provides a complete interface for algorithms and functions just as **GraphTool** would in most cases. However, because **GraphTool** is running a full graphical, mouse-oriented interface and the **Wrapper** has a textual interface, some commands function differently. These differences are documented along with the commands in the section below. In addition, the **Wrapper** can run both in interactive mode, where it will prompt you for input for certain commands, or in non-interactive mode, where it assumes that no user interaction is necessary for the algorithm to run and returns arbitrary values for commands that require input. Again, these differences are documented along with the commands themselves. Technical documentation for running the **Wrapper** is in an appendix to this document (section B).

Another way the **Wrapper** is different from **GraphTool** is in the way it handles the exiting of an algorithm. If the child exits on its own and returns an exit code, the **Wrapper** will exit with this return code. This enables you to check the exit code of the **Wrapper** in a shell script and take action on it. For example, if your algorithm can fail in certain cases, you can exit with some error code in those cases to indicate that it didn't do what it was supposed to. Then, when you are testing your algorithm, check the exit code of the **Wrapper** to see why your algorithm finished. When the child is terminated by a signal, the **Wrapper** will print the message `Child killed by SIGSEGV`, with whatever the actual signal was, on standard error and exit with a -1.

Finally, because the **Wrapper** doesn't have a window like **GraphTool**, the #display block may be destroyed or corrupted more easily. The **Wrapper** does set up a coordinate system with the same defaults as for **GraphTool**, but it is only set up once when the **Wrapper** is activated and cannot be reset without re-running the **Wrapper**. This may lead to an algorithm getting a coordinate system which is only a point or nonsense. You should be careful when using the window position and sizing parameters available in the **GraphTool** library (section 4.5) and in files (section C).

Note that although the **Wrapper** does not display the current graph, it still maintains the appearance attributes such as node colors (patterns in monochrome) and edge widths.

## 4.4 Interactive Communication with an Algorithm

An algorithm sends and receives commands through the standard input/output channels. This means that the standard printf/scanf and putchar/getchar commands can be used for all input and output rather than some exotic socket communication line or such. All commands are also human-readable ASCII lines, further enhancing the ease of programming and debugging. One thing to watch out for is that all commands must be terminated by a new-line and must be sent as a single entity to ensure complete transfer.

The first communication between a client and server is automatic and occurs at the time when the client is executed. The server sends the current graph to the algorithm on the algorithm's stdin in the same format as if it were saving it. (See section C.) The only difference is that the server terminates the input with <> to indicate that the graph data has stopped. EOF is not necessarily defined on the communication line so it is not a good indicator of the presence of data. A side effect of saving the graph is to renumber the graph from one so the algorithm is guaranteed a consecutive numbering system when dealing with its input. Once the initial graph has been consumed from the input, commands and replies can flow freely between the client and server.

The servers do very little error checking of commands so it is very important that they appear precisely as they are here. Specifically, the only checking that the servers do on the input is to check the numbers for the commands that refer to objects by their numbers. If a server receives a command using a number that refers to a non-existent object, the command is ignored. If debug mode is on, an error message will be logged into the debug file.

In the following description commands are typed as they would appear in `typewriter type` so you know exactly what **GraphTool** expects. The characters < and > enclose parameters and are not included in the file. A parameter is described with a type: int for integer, double for floating point and string for a string of characters. The type is followed by a name describing what should actually be typed for the parameter. The angle brackets in quotes indicate that they are actually part of the command to be sent.

`#append <string filename> <double x> <double y> <double width> <double height>`

For **GraphTool**, puts the given filename in the Filename item and appends a graph file to the current graph in the given area just as if you had typed the filename into the Filename item, clicked the Append button and selected the area. No error messages will be displayed, but an error file will be created if errors are found in the file.

For the **Wrapper**, appends the specified graph to the current graph. No errors will be displayed but an error file may be created.

`#circle <double x> <double y> <double r> <int op>`

**GraphTool** draws a circle of radius r at (x,y). Note that this is not something **GraphTool** remembers. It is the responsibility of the algorithm to clean up with a redraw. Otherwise, the circle will remain until the user does something so that **GraphTool** redraws the screen.

The x, y and r are specified in the graph's coordinate system.

The op specifies how the circle will be drawn. The codes are:

- 0 Draw
- 1 Clear
- 2 Exclusive OR

The **Wrapper** ignores this command.

`#clear`

Removes all nodes and edges from the current graph and clears the screen. Unlike the Clear editing command, nothing else, such as the coordinate system or the scrollbars, etc. is reset.

The **Wrapper** discards the current graph.

`#debug <string dump file>`

Starts debug mode processing. All commands that the server receives and all output that the server sends are dumped into the given file. Error messages will also be dumped here. The server appends to the given file so it is safe to reuse a file. The server also dumps the starting and ending times of the debug mode.

If an algorithm is terminated and debug mode is on, the server automatically closes the dump file.

`#delay <int milliseconds>`

After each command, **GraphTool** will delay this many milliseconds so that the execution speed of your algorithm can be controlled. Zero means no delay. Setting this value high (several seconds) will cause editor feedback delays.

The actual delay is usually close to the requested delay, but some machines do not provide support for this level of timing so the delay is approximated with a busy-loop. The approximation will be poor.

The **Wrapper** ignores this command.

`#dflush`

Causes the server to flush the debug file if the server is in debug mode. Because the debug file is buffered, there may be information lost in the buffer if the algorithm crashes when the debug file is open. To avoid this, the debug file can be flushed to write all material currently in the buffer to the file. This does take time, though, so this command should probably not be used in critical loops.

`#directed <int status>`

Changes the directed status of the graph to what is given. Any non-zero number for status will make the graph directed and a zero will make the graph undirected.

`#edge+ <int number> <int from> <int to> <double weight> <int width> <int color (style)>`

Adds an edge with the given values into the graph. Again, duplicate numbers are not an error. `From` and `to` are node number of nodes that exist already. If the from or to node does not exist, this command is disregarded. The color (styles in monochrome) are as described in section C.

`#edge- <int number>`

Removes an edge from the graph.

`#edgep <int number> <int width> <int color (style)>`

Sets the width and line color (style in monochrome) for an edge.

**#getdepth**

**GraphTool** returns the depth of the display. Color displays have a depth greater than one.

The **Wrapper** ignores this command.

**#gettext <‘<’string prompt‘>’> [<‘<’string default entry‘>’> ]**

**GraphTool** pops up a window with a text entry field in it. The prompt given is used as the prompt in the window for the text. An optional default entry will be placed in the field if specified. The <> are needed to enclose the strings so that spaces may be included.

In the **Wrapper**, if the **Wrapper** is in interactive mode, the user will be prompted for input with the given prompt and the default in brackets by it. If the user just hits return, the default is sent back. If the **Wrapper** is not in interactive mode, a warning message will be printed alerting the user that a text request was ignored and the **Wrapper** will send back an empty text return.

An example would be: **#gettext <Number of layers?> <10>**

There is no way to specify a string with a > in it, although < can be used.

The server returns the input in a command of the form:

**#text <string>**

The <> are not included here. Also, in **GraphTool**, the user can cancel the input, in which case **GraphTool** returns only #text, or the user can input nothing, in which case there will be one space after #text. Under the **Wrapper**, there is no way to enter nothing unless the default is empty also. If the **Wrapper** receives a #gettext in non-interactive mode, it will return a #text with nothing, as if the #gettext had been cancelled.

**#line <double x1> <double y1> <double x2> <double y2> <int op>**

**GraphTool** draws a line from (x1,y1) to (x2,y2). #line functions in the same manner as #circle.

The **Wrapper** ignores this command.

**#load <string filename>**

In **GraphTool**, puts the filename in the Filename item and loads a graph file just as if you had typed the given filename in the Filename item and clicked on the Load button. No errors will be displayed, although an error file will be created if errors are found in the file. No confirmation of this action will be requested if the current graph is unsaved, so be careful not to issue this command unless you know the current graph can be discarded.

The **Wrapper** loads the specified graph. No errors will be displayed but an error file may be created.

`#ltext <string text> <double x> <double y> <int op>`

> Draws the text between the <> at the given position with the given operation. The (x,y) is the left edge and baseline position for the text. #ltext functions in the same manner as #circle.
>
> The **Wrapper** ignores this command.

`#msg <string message>`

> In **GraphTool**, displays `message` in the Message field.
>
> In the **Wrapper**, depending on what mode you have run it in, the message may be logged to a file, printed on the standard output or ignored.

`#node+ <int number> <double x> <double y> <double weight> <int color>`

> Adds a node with the given values into the graph. Duplicate numbers are not an error; the first node with the number will be used in later references to the number. The colors (patterns in monochrome) are the same as described in section C.

`#node- <int number>`

> Deletes the node with the given number.

`#node= <int number> <double x> <double y>`

> Moves an existing node to new coordinates.

`#nodebug`

> Turns off debug mode and closes the debug file. If debug mode was not active, this does nothing, so it is always a safe command.

`#nodep <int number> <int color (pattern)>`

> Sets the fill color (pattern in monochrome) for a node.

`#noedges`

> Removes all edges from the current graph.

`#plain`

> Performs the same function as the Plain button.

`#rect <double ulx> <double uly> <double lrx> <double lry> <int op>`

> **GraphTool** draws a rectangle with upper-left point (ulx,uly) and lower-right point (lrx,lry). #rect functions in the same manner as #circle.
>
> The **Wrapper** ignores this command.

`#redraw`

Redraws the current graph. The screen is erased before the redraw, so this can be used to remove anything the algorithm has drawn that is no longer needed.

The **Wrapper** ignores this command.

`#renumbere <int start>`

Renumbers the edge list starting from the given number.

`#renumbern <int start>`

Renumbers the node list starting from the given number. In the **Wrapper**, the edge list is modified to reflect the new node numbers for from and to.

`#select-e`

Causes **GraphTool** to enter EDGE mode and wait for the user to select an edge.

If the **Wrapper** is in interactive mode, the user will be prompted for an edge number. Otherwise, a warning is printed and a garbage edge is sent to the algorithm. The edge will have a number of -1, so you can check for this in your algorithm if you think it will be necessary.

Once the user has selected an edge, the server sends it to the algorithm in the form:

```
#edge <int number> <int from> <int to> <double weight> <int width>
      <int color (style)>
```

Note that the server just sends the information to the algorithm and then goes back to doing nothing. The algorithm can do anything it wants after it issues a #select command, including not reading its input for quite some time after it actually receives the edge notification. So it is quite possible for the server to be waiting for a selection while the algorithm is doing something else. The algorithm would then read its input when it was ready.

`#select-n`

Causes **GraphTool** to enter NODE mode and wait for the user to select a node.

The **Wrapper** handles #select-n just like #select-e.

Once the user has selected a node, the server sends it to the algorithm in the form:

```
#node <int number> <double x> <double y> <double weight> <int color
      (pattern)>
```

#select-n returns its value in the same manner as #select-e.

`#sendgraph`

Causes the server to send the current graph to the algorithm just as it did automatically when the algorithm was started. The graph is renumbered from one when it is sent so that the numbering system is coherent.

`#sync`

Allows synchronization of the server and the algorithm. The server replies with the line:

<>

This command is not strictly necessary but it ensures that the server receives all of the input from the algorithm. If the algorithm exits with commands still making their way between the server and the algorithm, they will be ignored. So an algorithm that wants to make sure all of its command are sent will send #sync as the last command and then wait for the <>. Because the server won't send the <> until it gets the #sync, you are assured that the server has also seen any commands that preceded the #sync. The algorithm can then safely exit. If the algorithm's last action was to wait on something, such as #select-n, #select-e or #waitclick, #sync is not necessary because the algorithm will already have waited for the server to synchronize with the algorithm and send the appropriate information.

#sync's only action is to cause the server to send <>, so it can be used any number of times at any time during the execution of an algorithm as long as you remember to read the <> in to clear it.

`#waitclick`

Causes **GraphTool** to enter WAIT mode and wait for the user to click a mouse button.

If the **Wrapper** is in interactive mode, it will prompt the user for a button number. If the **Wrapper** is not in interactive mode, -1 is sent for the button number.

Once the user has selected a mouse button, the server sends the click to the algorithm in the form:

**`#click`** <**`int button`**> The buttons are:

1. Left
2. Middle
3. Right

#waitclick returns its value in the same manner as #select-e.

`#weight`

Performs the same function as the Weight button.

### 4.5 The GraphTool Library

A library of useful subroutines has been written for writers of graph algorithms. The only current language bindings are in C. The library attempts to define function prototypes as in the ANSI standard. Your compiler may not be able to handle these prototypes, in which case you must change a define in the library header file called ARGS to () so that arguments are not declared.

Data structures for nodes and edges are defined as follows:

```
struct edge {
  struct edge *from_next,*from_prev;
  struct edge *to_next,*to_prev;
  int num;
  int i_from,i_to;
  struct node *p_from,*p_to;
  double weight;
  int width,style;
  double from_angle,to_angle;
  void *data;
};

struct node {
  struct node *next,*prev;
  int num;
  struct edge *vh,*vt;
  double x,y,weight;
  int pattern;
  void *data;
};
```

The *data fields are unused by the library and are present to allow you to hook any private data you wish onto edges and nodes.

The library's graph reading functions set a number of global variables for your use:

`int version`

The file format version number of the graph read in.

`int num_nodes,num_edges`

The number of nodes and edges read in.

`double x_off,y_off,x_scl,y_scl`

`double zoom_multx,zoom_multy`

`int directed`

`int node_radius`

`int wx,wy,ww,wh`

`int zoom_level`

`int x_scroll,y_scroll,x_scroll_pos,y_scroll_pos,x_scroll_len, y_scroll_len`

A complete description of what these variables mean is in section C.

The library contains the following macros:

`line_buffer(FILE *for)`

line_buffer is a macro that sets the given file pointer into a line-buffered mode. It is present because this is done differently on different systems, and if you use it you can port your code without changes because the **GraphTool** library should have the appropriate definition on the new system. If line_buffer fails, you may have to edit the library header file to change it to something that is correct for your system.

The next group of macros exists to deal with vertex edge lists, (generated and sorted by vertex_edge_lists and sort_vertex_edge_lists, described later), which do not have a regular format. It is strongly recommended that you use these macros to traverse vertex edge lists rather than your own code, as the interconnections between vertex edge lists can become complex. Using the macros will also shield you from any changes in the lists' representation.

These macros come in various forms depending on the type of the arguments and the type of the result coming back. Most of them take two arguments: an edge and a vertex. The edge is always a struct edge *. The following naming convention is used: if the vertex is an index (number), append an I to the macro's base name. If the vertex is a struct node *, append a P. If you want an index back, append an I to what you just appended. If you want a pointer back, append a P. So OPP_VERTIP(edge,vertex) takes an edge pointer and a node number and returns an edge pointer.

If you are confident that you will never get a null pointer for SUCC, PRED, etc., you can reference directly from them, as in SUCC(edge,vertex)->num.

Note that because these are macros, the arguments are textually substituted and may occur more than once in the final code. Therefore, do not use arguments with side effects, like OPP_VERTIP(e++,vertex) because the e++ will be evaluated twice and the result will not be what you expected.

`ANGLE(edge,vertex)`

Returns the angle the given edge makes from the vertex to its opposite vertex. This is only valid on a sorted edge list.

P and I forms.

`CCW(edge,vertex)`

Returns the next edge in the counter-clockwise direction in the given vertex's sorted edge list. Sorted edge lists are circular, so this will never return NULL.

PP, PI, IP, II forms.

`CW(edge,vertex)`

Returns the next edge in the clockwise direction in the given vertex's sorted edge list. Sorted edge lists are circular, so this will never return NULL.

PP, PI, IP, II forms.

`ENUM(edge)`

Returns either the number of the edge or -1 if the edge pointer is NULL. Useful in combination with SUCC and PRED to simulate PI and II forms.

`OPP_VERT(edge,vertex)`

Returns the vertex opposite the given vertex on the given edge.

PP, PI, IP, II forms.

`PRED(edge,vertex)`

Returns the previous edge in the given vertex's edge list.

PP, IP forms.

`SUCC(edge,vertex)`

Returns the next edge in the given vertex's edge list.

PP, IP forms.

The library contains the following functions:

`void ar_read_graph(FILE *from, struct node **nd_ar,struct edge **ed_ar,int nd_max, int nd_start,int ed_max,int ed_start)`

Reads a graph from the given file pointer into an array. If nd_ar or ed_ar is NULL, an array of *_max+*_start elements is malloc'ed for you and the *_ar pointer is set appropriately. Nodes and edges are read into the array starting at the starting element given. The next and prev pointers are set correctly so that the array can also be treated as a doubly linked list. The edge list uses the from_* pointers.

Note that giving a starting position does not affect the numbering of the graph; it only affects the position in the array in which the graph will be stored.

`int click_message(printf arguments)`

Takes the same arguments as a printf call and sends the resulting string as a #msg command. If the message is not terminated by a newline, one is added automatically.

Click_message then does a #waitclick and scans the input for the #click return. Any floating newlines in the input are discarded so click_message will leave a clean input pipe after it returns. Click_message sends a #dflush before it waits so that the debug channel is flushed. Finally, click_message returns the number of the button pressed.

`void edge_chk_end(struct node *node,struct edge *edge)`

Links the given edge into the given node's vertex edge list along the from_* pointers or to_* pointers depending on whether the node is the from node or to node.

`struct edge *edge_num_ptr(struct edge *head,int n)`

Returns a pointer to the n'th edges from the given list head. If the list ends before the n'th edge, NULL is returned. The from_next pointer chain is followed. The first edge is number one.

```
void edge_weight_matrix(struct edge *head,double **matrix,int n,int
                        directed, double initial,int boolean)
```

Creates a matrix where entry (row,column) contains the weight of the edge with fields (from,to)=(row,column). If matrix is NULL, a matrix of size [n+1][n+1] is malloc'ed for you. Note that n is the number of nodes, with the nodes considered to start at one, not zero, so everything is n+1 elements in size.

The matrix is filled with the given initial value and then all edges are put in. For undirected graphs, both (from,to) and (to,from) entries are set for a particular edge, while only (from,to) is set for a directed edge.

If boolean is non-zero, non-zero weights are put in the matrix as one, so with an initial value of zero a boolean matrix is created.

The edge list is traversed along the from_* pointers.

```
void fill_edge(struct edge *fill, int number,struct node *head,int
          from,int to,double weight, int width,int style)
```

Fills the given edge's fields with the given information. Sets the p_from and p_to fields with node_num_ptr using the given node list head. From_next, from_prev, to_next, to_prev and data are set to NULL.

```
void fill_node(struct node *fill, int number,double x,double y,double
          weight,int pattern)
```

Fills the given node's fields with the given information. Sets next, prev, vh, vt and data to NULL.

`void flush_pipe()`

Ensures that algorithm communication pipes are flushed by sending #sync and then waiting for the <>. All input from the server is ignored until the <> is sent, so the input pipe is emptied up to the <>. #dflush is sent before the wait so that the debug channel is flushed.

`void free_edge_list(struct edge *edges)`

Frees all of the edges in the edge list. The list is traversed along the from_* pointers.

`void free_node_list(struct node *nodes)`

Frees all of the nodes in the node list and their edge lists. The edge lists may be circular, as after a sort_vertex_edge_lists, or normal, as after any of the vertex edge list creation routines have acted on a node.

`void ll_read_graph(FILE *from,struct node **nd_head, struct edge **ed_head)`

Reads a graph from the given file pointer into a linked list. The nd_head and ed_head parameters are set to point to the heads of the node and edge lists, respectively. The edge list is constructed along the from_* pointers.

`void message(printf arguments)`

Takes the same arguments as a printf call and sends the resulting string as a #msg command. If the message is not terminated by a newline, one is added automatically. Message sends a #dflush before it waits so that the debug channel is flushed.

`void new_vertex_edge(struct node *from,struct node *to, struct edge *edge)`

Allocates a new edge, copies the given edge's information into it and links the new edge into both the from and to node's vertex edge lists along the proper pointer chain.

`void new_vertex_edge_lists(struct edge *head)`

For each vertex, creates a list of the edges that impinge upon that vertex using the vh (head) and vt (tail) pointers. Edges are added to both the from node and the to node along their respective pointers. Therefore, when traversing an edge list from the vh pointer of a node, if the edge's from node is the same as the node you started from, use the from_* pointers. If the edge's from node is not the node you started from, use the to_* pointers. The SUCC and PRED macros described above shield you from this problem, so use them whenever you can.

The list is unsorted and is new. The old edge list is undisturbed.

`struct node *node_num_ptr(struct node *head,int n)`

Returns a pointer to the n'th node from the given list head. If the list ends before the n'th node, NULL is returned. The first node is number one.

`void number_from(struct node *nodes,int nstart,struct edge *edges,int estart)`

Renumbers the node and edge lists starting at the given starting value for the head and increasing along the list. The edge list is followed along the from_* pointers. The i_* fields in the edge list are updated to reflect the new node numbers. Vertex edge lists are NOT updated.

```
void read_graph(char *name,FILE *from, FILE *error,void
            (*add_node)(),void (*add_edge)(),int header)
```

Reads a graph from the given file name or pointer. If the from file pointer is NULL, an attempt to open 'name' is made. The error file pointer is a single file that any input errors will be logged to. If error is passed as NULL, 'name.err' is used. If name is NULL, no error file is kept.

The directed global is set if header is non-zero and a #graph block is encountered. The #graph block is mandatory, so directed will be set for all but malformed files. If a #display block is present in the file and header is non-zero, the other globals will be set with its values. The #display block is optional. Therefore, if you want to hand- craft a file and don't have any idea as to what to put in the #display block, leave it out. #include links are followed.

For each node and edge, the given add function is called. The function pointer may be NULL if you don't want additions of that type. The format for a node add function is:

```
void add_node(int number,double x,double y, double weight,int pattern)
```

The number given is the node's number in the current numbering scheme kept by read_file. You may ignore it if you wish. The other arguments contain the same data as in the #node+ command.

An edge add function looks like:

```
void add_edge(int number,int from,int to, double weight,int width,int
          style)
```

The number given is the node's number in the current numbering scheme kept by read_file. You may ignore it if you wish. The other arguments contain the same data as in the #edge+ command.

```
void skip_graph(FILE *from)
```

Sets num_nodes and num_edges to zero and then calls read_graph(NULL, from, NULL, NULL, NULL) to set the rest of the globals but ignore all node and edge information.

```
void sort_vertex_edge_lists(struct node *head)
```

Sorts the vertex edge lists into angular order. The CW and CCW macros described above allow you to traverse the resulting list in clockwise and counter-clockwise order.

This function traverses and sorts lists of the the type generated by vertex_edge_lists using the vh and vt pointers in the nodes, if you wish to construct your own lists for sorting.

`void vertex_edge(struct node *from,struct node *to, struct edge *edge)`

Links the given edge into both the from and to node's vertex edge lists along the proper pointer chain.

`void vertex_edge_lists(struct edge *head)`

Performs the same function as new_vertex_edge_lists, but uses the old list as storage space instead of allocating new space. The old edge list is destroyed in the process and cannot be used again.

`void write_graph(FILE *to,int header,struct node * nodes, struct edge *edges)`

Writes the given graph to the given file pointer in the **GraphTool** file format. The edges list is followed along the from_* pointers. The graph is renumbered from one.

If header is non-zero, a #display header is written using the library's global variables. If you do not want a header or the global variables are not set, pass a zero for header and a header will not be written. As mentioned above, read_graph will read files with and without #display blocks.

## 4.6 Some Advice on Programming Algorithms

Use as many of the **GraphTool** library's routines as possible, as this will enable you to move your algorithms between different servers and server versions with a minimum of code changes.

Watch out for newlines in the communication streams. If you forget a newline somewhere the results can be disastrous. For example, when using a #select or #waitclick, the returned value includes a newline. Using scanf to pull the fields out of the input stream will work, but will leave the newline in the input channel, which can get in the way if you later need to read a line from the input. If you have more than one scanf in a row, things will work because most of the scanf scan types skip whitespace and so will skip the extra newlines. For something like a #text return, though, you can't just scan a string from the input because the return may contain spaces. You must scan the whole line from the input. When doing so, either avoid whitespace yourself or eliminate it at the other steps. Click_message and flush_pipe clear the input line for you, so if you can structure your code to let them do the work, good for you.

Know what I/O channels do what and can be used for what. Stdin is dedicated to receiving messages from the server, so you shouldn't expect any other input from it as you normally would. Stdout is dedicated to sending messages to the server, so *don't output anything else to it.* The easiest way to break your algorithm is to accidentally use a printf when you want to print out some message. Stderr is what you want to use. The stderr of an algorithm under **GraphTool** is the same as the stderr of **GraphTool**, which is usally on the tty that **GraphTool** was executed from, although it can be redirected. The **Wrapper** allows you to redirect the stderr of the algorithm to anything. This makes a really neat trick possible: when writing an algorithm, send all your debugging output to stderr. When you run the algorithm under **GraphTool**, you can see all of this on the tty you ran **GraphTool** on. When you want to run it under the **Wrapper**, but you don't want to see the debugging stuff, you can tell the **Wrapper** to send stderr to /dev/null.

### 4.7 An Example Algorithm

What follows is the actual C code for an algorithm that computes shortest paths using Dijkstra's shortest path algorithm. [1] The user is prompted for a from and a to node and the algorithm highlights the shortest path between the two with a thick, dashed line.

```
#include <stdio.h>
#include <values.h>
#include "graphtool.h"

#define NUM_NODES 300
#define NUM_EDGES 600
extern int num_edges,num_nodes,directed;
struct node *nodes=NULL;
struct edge *edges=NULL;
double dist[NUM_NODES][NUM_NODES],*distmat=dist;

main()
{
  double x,y;
  int from,to,i,j,cost[NUM_NODES],final[NUM_NODES],
      prev[NUM_NODES],min_cost,not_final,min_node;
  char junk[80];

  /* GraphTool passes the current graph as input
     when the process is activated */
  ar_read_graph(stdin,&nodes,&edges,NUM_NODES,1,NUM_EDGES,1);
  /* This line is usually critical.  If the stream being
     written to is block-buffered (the default) the
     stdio library will buffer characters and your
     commands won't appear as you would expect.
     Because commands are lines, line buffering
     is optimal */
  line_buffer(stdout);

  /* Your program goes here */
```

[1] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

```
  for (;;)
    {
      printf("#plain\n");

      printf("#msg Path from?\n");
      printf("#select-n\n");
      scanf("%s%d%lf%lf%lf%d",junk,&from,&x,&y,&x,&i);
      printf("#msg To?\n");
      printf("#select-n\n");
      scanf("%s%d%lf%lf%lf%d",junk,&to,&x,&y,&x,&i);

      /*
        A simple shortest path algorithm.
        Assumes a path exists
        */

      /* Initialize costs to `infinity' */
#define INFINITY MAXINT

      for (i=1;i<=num_nodes;i++)
        {
          cost[i]=INFINITY;
          final[i]=0;
          prev[i]=from;
        }

      /* Set initial costs */
      edge_weight_matrix(edges+1,&distmat,NUM_NODES-1,directed,INFINITY,0);
      for (i=1;i<=num_edges;i++)
        {
          if (edges[i].i_from==from)
            cost[edges[i].i_to]=edges[i].weight;
          if (!directed && edges[i].i_to==from)
            cost[edges[i].i_from]=edges[i].weight;
        }

      final[from]=1;
      not_final=num_nodes-1;
      for (;not_final;not_final--)
        {
          min_cost=INFINITY;
          for (i=1;i<=num_nodes;i++)
```

```
          if (!final[i] && cost[i]<min_cost)
            {
              min_node=i;
              min_cost=cost[i];
            }
        if (min_node==to) break;
        final[min_node]=1;
        for (i=1;i<=num_nodes;i++)
          if (!final[i])
            if (cost[i]>cost[min_node]+dist[min_node][i])
              {
                prev[i]=min_node;
                cost[i]=cost[min_node]+dist[min_node][i];
              }
      }
    for (i=to;i!=from;i=prev[i])
      for (j=1;j<=num_edges;j++)
        if (edges[j].i_from==prev[i] && edges[j].i_to==i)
          {
            printf("#edgep %d 5 4\n",j);
            break;
          }
        else if (!directed && edges[j].i_to==prev[i] && edges[j].i_from==i)
          {
            printf("#edgep %d 5 4\n",j);
            break;
          }

    i=click_message("Left - More, Other - Quit\n");
    if (i!=1) break;
  }
 flush_pipe();
}
```

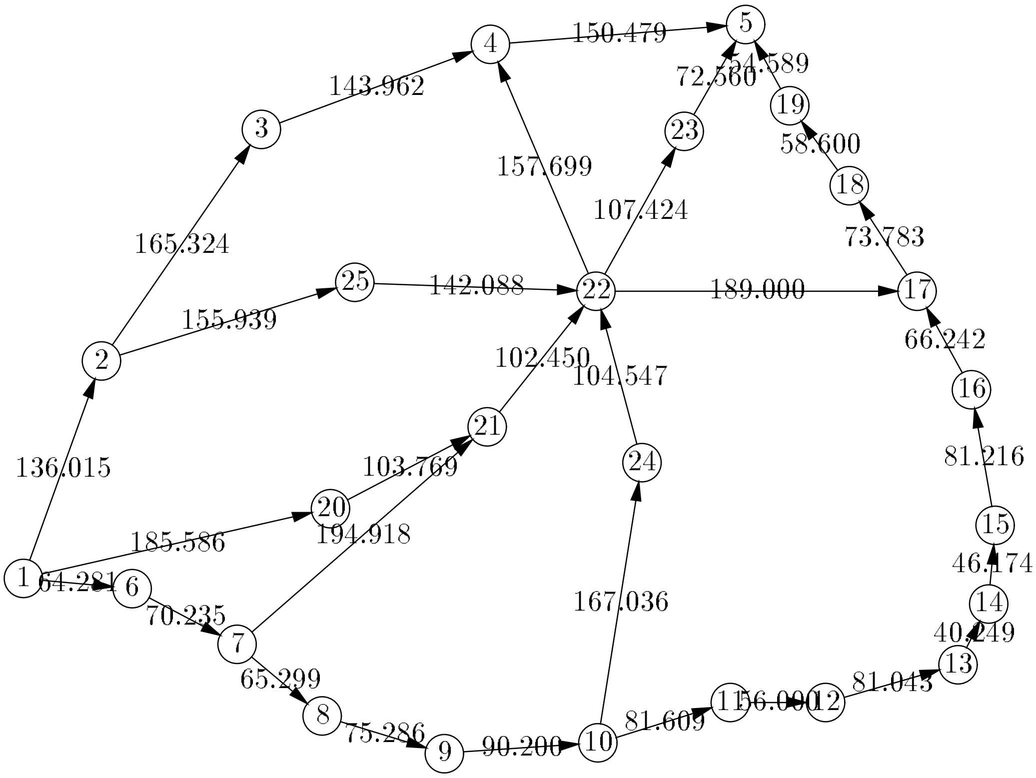


Figure 2: The Original Graph with Edge Weights Shown

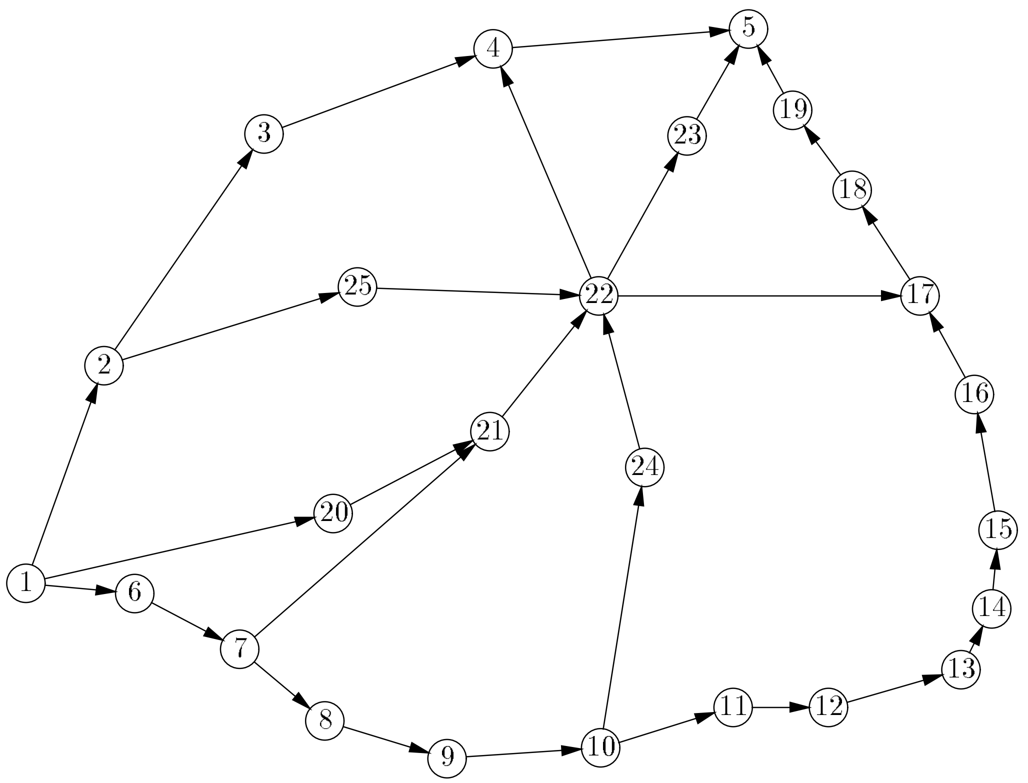

Figure 3: The Original Graph

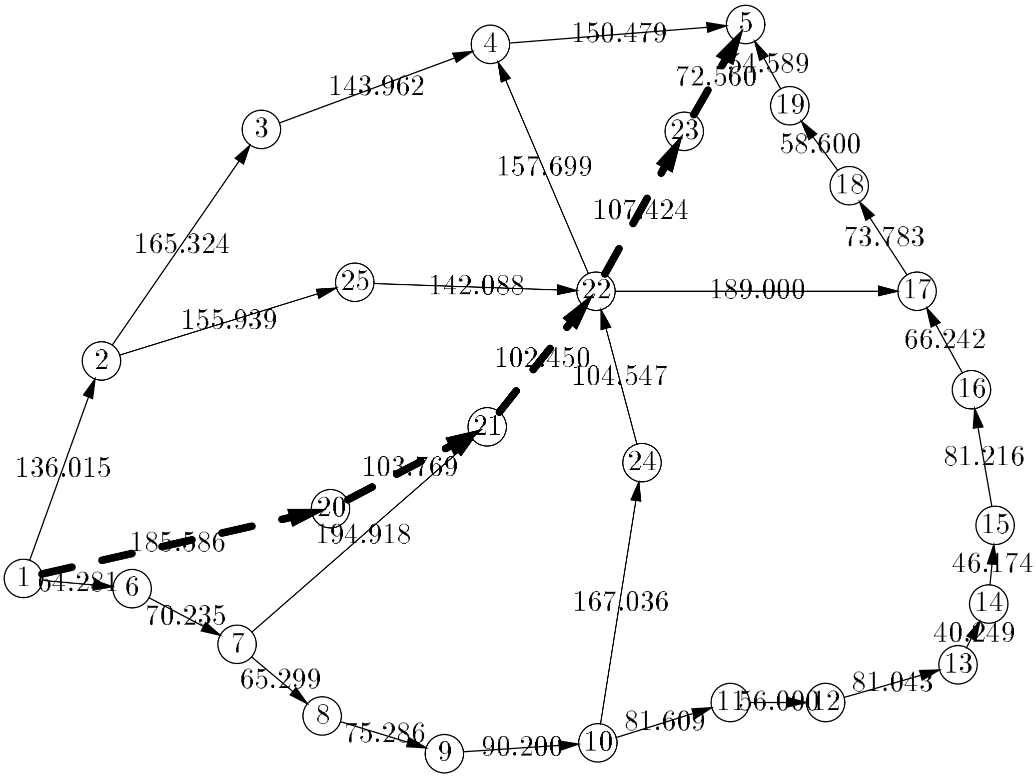


Figure 4: The Shortest Path Between Nodes 1 and 5 with Edge Weights Shown

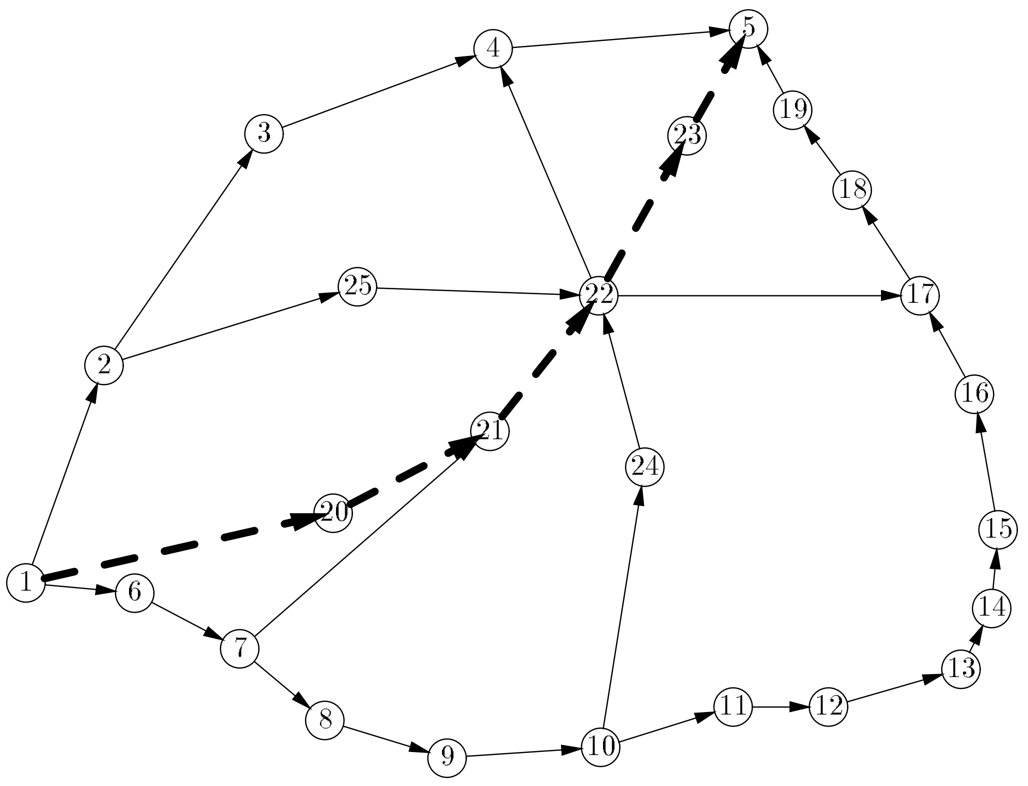


Figure 5: The Shortest Path Between Nodes 1 and 5

# A GraphTool

**GraphTool** takes the following command line arguments:

| | |
|---|---|
| `-display` | X display. |
| `-geometry` \| = | Geometry of canvas window. |
| `-background` \| `-bg` | Canvas window background color. |
| `-foreground` \| `-fg` | Canvas window foreground color. |
| `-borderwidth` \| `-bw` | Canvas window border width. |
| `-font` \| `-fn` | Panel window font. |
| `-name` | Resource retrieval instance name. |
| `-xrm` | Define a resource |

An additional argument is assumed to be an initial file name and sets the Filename item to the argument. The file is not loaded, though.

**GraphTool** searches the system databases for the following resources:

| | |
|---|---|
| `display (Class Display)` | X display. |
| `name (Class Name)` | Resource retrieval instance name. |
| `geometry (Class Geometry)` | Canvas window geometry. |
| `background (Class Background)` | Canvas window background color. |
| `foreground (Class Foreground)` | Canvas window foreground color. |
| `borderWidth (Class BorderWidth)` | Canvas window border width. |
| `font (Class Font)` | Panel window font. |

## B The Wrapper

The **Wrapper** is a small program which furnishes a completely **GraphTool** compatible harness for **GraphTool** algorithms. It is mainly intended for running an algorithm in a batch mode, but is not limited to doing so.

Upon startup, the **Wrapper** reads a graph from a file or its standard input and passes the graph to the algorithm being run. The algorithm can then perform all its normal actions and the **Wrapper** will track the current state of the graph. When the algorithm exits, the **Wrapper** writes the current graph to a file or its standard output.

When using the standard input and output for graph I/O, you should be careful not to use interactive mode because interactive mode uses stdin for its input. If you have already dedicated stdin to a graph, you are out of luck. The **Wrapper** provides complete control over the graph input file, output file and stderr, so there shouldn't be any difficulties.

The **Wrapper** is intended to be run from the UNIX command line. Its invocation takes the form:

```
wrapper [-i] [-t] [(+|-o) file] [(+|-)e file]
        [-gi file] [(+|-)go file] [-ngi] [-ngo]
        algorithm [algorithm args]
```

`-i` Puts the **Wrapper** in interactive mode. In interactive mode, the user is prompted for input on things where input is necessary such as a select or a waitclick. If these commands are issued when the **Wrapper** is not interactive, garbage comes back, which could cause problems.

`-t` Asks the **Wrapper** to time the execution of the algorithm. The time is taken just before the initial graph is sent and just after the child exits. It may include some operating system overhead time to set up the algorithm and shut it down, but in all but the most pathological cases this shouldn't strongly affect the overall time.

The run time is printed as the last output from the **Wrapper** in the form `Running time: 00:00:00.000000`. The accuracy of the fractional portion of the time is limited by the system and probably will never be trustworthy beyond milliseconds.

`(+|-)o file` Directs the **Wrapper** to send the text from all #msg commands to the given file. If -o is used, the file is opened for writing and any previous contents it had will be destroyed. If +o is used, the file is appended to so multiple runs will accumulate.

`(+|-)e file` Directs the **Wrapper** to send all algorithm output on the standard error to the given file. The file is overwritten or appended to according to whether - or + is used.

| | |
|---|---|
| | This is very useful for collecting results over a batch session through the standard error while also allowing results to be viewed on the tty when the algorithm is run under **GraphTool**. |
| `-gi file` | Causes the **Wrapper** to read its initial graph from the given file. The .g is not automatically appended. If this option is not present, the **Wrapper** reads from its standard input. |
| `(+\|-)go file` | Causes the **Wrapper** to write the final graph to the given file. The .g is not automatically appended. If this option is not present, the **Wrapper** writes to its standard output. The file with be overwritten or appended to according to whether - or + is used. |
| `-ngi` | Tells the **Wrapper** to send an empty graph to the algorithm. **GraphTool**'s default coordinate system is used in the #display block. The **Wrapper** does not read in an initial graph. |
| `-ngo` | Tells the **Wrapper** not to write the final graph. |
| `algorithm` | Specifies the path of the algorithm to be run. |
| `algorithm args` | The remaining arguments to the **Wrapper** are passed to the algorithm as its command line so that command line switches may be passed to the algorithm itself as in **GraphTool**. |

## B.1 Examples Using the Wrapper

The simplest use of the **Wrapper** is in a standard UNIX pipeline. If we had an algorithm which generated a random graph with the number of points specified on the command line and an algorithm which computed a convex hull for a graph, we could easily put them into a pipeline:

```
% wrapper -ngi random_graph 50 | wrapper convex_hull
```

random_graph will generate a fifty vertex random graph in its **Wrapper** and the **Wrapper** will send the graph on its stdout when random_graph exits. This travels through the pipe to the convex_hull **Wrapper** which gives it as input to convex_hull. Convex_hull computes the convex hull and the final graph is printed out on the tty, or it could be redirected into a file.

If you have an algorithm which takes input, like the shortest path algorithm from above, you can't just run it in a pipeline like the previous example because the interactive mode takes its input from stdin. You would do this:

```
% wrapper -ngi -go /tmp/rg random_graph 50
% wrapper -i -gi /tmp/rg shortest_path
```

The first **Wrapper** writes its graph to /tmp/rg rather than stdout, (which could have also been accomplished with redirection) and the second **Wrapper** takes this graph as input. This must be specified with -gi instead of redirection so that stdin is available for typed input.

The **Wrapper** is run in interactive mode, so you will be asked to type in the numbers of the from and to nodes on the tty and also the waitclick number. The final graph will then go out on the stdout.

For more complex things, like testing, shell scripts are best. For example, suppose you had an algorithm that, given a point set, finds the number of triangles that can be made with three of the points that don't contain any of the other points. You want to run this many times to empirically determine the relationship between the number of points and the number of empty triangles. You have written the algorithm so that it writes the number of empty triangles to its stderr; exits with zero if successful and exits with one if it failed for any reason. The following script will perform the tests for you:

```
#! /bin/csh -f

# Usage: <script name> [number of points [number of iterations]]

# Initialize number of points and iterations from command line
# Default to fifty points and infinite iterations

if ($#argv > 0) then
  set points = $1
else
  set points = 50
endif

if ($#argv > 1) then
  set iter_max = $2
else
  set iter_max = -1
endif

set iter = 0

while (1)

# Are we finished?
  if ($iter_max > 0 && $iter >= $iter_max) break

# Write out iteration number
  @ iter ++
  echo -n $iter "- "

# Generate graph
  wrapper -ngi -go /tmp/empty$$ random_graph $points
```

```
# Run algorithm on graph
  if ( { wrapper -gi /tmp/empty$$ -ngo -e results.1 empty_triangles } ) then
# Print out iteration and results
    echo $iter - `cat results.1` empty triangles
# Log results on end of results file and delete single result for next time
    cat results.1 >> results
    /bin/rm -f results.1
  endif
end
# Remove temp file
/bin/rm -f /tmp/empty$$
```

## C File Format

Files are line-formatted, i.e., an entire line is read in at once when parsing, so only one keyword can be present on a line. Keywords begin most lines and tell **GraphTool** what to do with the rest of the line. It is very important to type the commands in exactly the way they are presented here. **GraphTool** can catch most errors but a single simple spacing error can cause the rest of the file load to fail. The keywords and lines may appear in any order in the file separated by any number of blank lines.

The very first line of the file must be:

`int version_number`

A number identifying the file format version.

### C.0.1 Keywords

`#include <string filename>`

This command causes the file `filename` to be parsed in the same way as the current file is being parsed. Additional headers are ignored and a warning is given if one is found. The first included file's name is remembered and written to a saved file in lieu of a header block so that standard, shared headers can be preserved automatically across editing sessions.

`-- <comment>`

If the first two characters of a keyword are −−, the line is ignored. Note that this commenting only works at the keyword level; it is not possible to include free-form comments. That means you can't insert comments in a block like #display or #graph. However, nodes and edges are not keywords, so it is possible to insert comment lines between lines defining nodes or edges.

`#display`

Introduces a display header block. The header is:

`int window x (wx), int window y (wy), int window width (ww), int window height (wh)`

Defines the canvas window.

`double x_offset, double y_offset double x_scale, double y_scale`

Defines the coordinate mapping from canvas to graph by:

$$canvas\_x = (int)((graph\_x - x\_offset)/x\_scale)$$

Similarly for y.

`int zoom_level`

A power of two indicating the current zoom level.

`double zoom_multx, zoom_multy`
: Scaling factors that control zooming.

`int x_scroll, y_scroll`
: Offsets that control the position of the current graph.

`int x_scroll_pos, y_scroll_pos, x_scroll_len, y_scroll_len`
: Values controlling the current appearance of the scrollbars.

`int node_radius`
: Defines the radius of the node circles in pixels.

The header values may not be on the same line as #display. They may be on any lines thereafter. All values must be present.

`#graph`

Introduces a graph header block. The header is:

`int directed`
: 1 = Directed edges, 0 = undirected.

`#nodes`

Signifies that any non-blank, non-comment lines following this line should be treated as node definitions. #nodes must be on a line by itself.

`#edges`

Similar to #nodes, but for edges.

### C.0.2 Nodes and Edges

Input lines that do not have one of the previously described keywords are interpreted according to the node/edge flag. The node/edge flag defaults to nodes.

A node is (must be on one line):

`double x, double y, double weight, int pattern`

The colors (patterns in monochrome) are:

- -3 Hollow
- -2 Empty
- -1 Solid
- 0 Blue (17% Grey)
- 1 Green (20% Grey)
- 2 Green Yellow (25% Grey)

- 3 Yellow Green (50% Grey)
- 4 Yellow (75% Grey)
- 5 Orange (80% Grey)
- 6 Orange Red (83% Grey)
- 7 Red (Root Grey, essentially a different 50% grey)
- 8 Violet Red (Not available)
- 9 Violet (Not available)
- 10 Blue Violet (Not available)

An edge is (must be on one line):

```
int from_node_number, int to_node_number, double weight, int width, int
style
```

The width can be any number greater than or equal to zero. The colors (styles in monochrome) are:

- -1 Solid
- 0 Blue (Dashed)
- 1 Green (Dotted)
- 2 Green Yellow (Dash-Dot)
- 3 Yellow Green (Dash-Dot-Dot)
- 4 Yellow (Long dashes)
- 5 Orange (Not available)
- 6 Orange Red (Not available)
- 7 Red (Not available)
- 8 Violet Red (Not available)
- 9 Violet (Not available)
- 10 Blue Violet (Not available)

Nodes and edges are numbered automatically as they are read in, starting at one. Therefore, the from and to node numbers in the edge lists must be set up as if the nodes were numbered from one. When saving from **GraphTool**, this is done automatically.

### C.0.3 An Example File

```
100
#display
20 20 1000 800
0.000000 799.000000 1.000000 -1.000000
1 1.000000 1.000000 0 0
0 0 1000 800
10
#graph
1

#nodes
302.000000 403.000000 1.000000 -3
440.000000 507.000000 1.000000 -3
527.000000 392.000000 1.000000 -3
615.000000 508.000000 1.000000 -3
737.000000 387.000000 1.000000 -3
619.000000 263.000000 1.000000 -3
432.000000 257.000000 1.000000 -3
213.000000 592.000000 1.000000 -3
836.000000 608.000000 1.000000 -3
827.000000 158.000000 1.000000 -3
201.000000 150.000000 1.000000 -3

#edges
1 2 1.000000 1 -1
2 3 1.000000 1 -1
3 4 1.000000 1 -1
4 5 1.000000 1 -1
5 6 1.000000 1 -1
6 3 1.000000 1 -1
3 7 1.000000 1 -1
7 1.000000 1 1 -1
11 8 1.000000 1 -1
8 9 1.000000 1 -1
9 10 1.000000 1 -1
10 11 1.000000 1 -1
```