GRAPHSHOP: AN INTERACTIVE SOFTWARE ENVIRONMENT FOR GRAPH THEORY RESEARCH AND APPLICATIONS

by

Aaron Andersen

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Mathematics

Approved:

David E. Brown Major Professor	Minghui Jiang Committee Member
LeRoy B. Beasley Committee Member	Byron R. Burnham Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2011

Grahpshop Copyright © Aaron Andersen 2011

Software source code available under the GNU General Public License

Documentation and references available under the Creative Commons Attribution-ShareAlike 3.0 Unported License

The Qt framework was obtained under the GNU Lesser General Public License

Full copies of the GPL, LGPL, and CC-BY-SA licenses are referrenced in the Bibliography.

ABSTRACT

GraphShop: An interactive Software Environment for Graph Theory Research and Applications

by

Aaron Andersen, Master of Science

Utah State University, 2011

Major Professor: Dr. David E. Brown
Department: Mathematics

Graph Theory is the mathematical study of the structure of abstract relationships between objects. Although these constructions (graphs) are themselves purely theoretical, their ability to model pair-wise relationships in systems of arbitrary complexity yields abundant direct correspondence with numerous important physical and societal systems in the real world. Additionally, the simple discrete nature of fundamental graph structures allows for easy pseudo-geometric visualization of graphs in a wide variety of ways. Taken together, these two properties suggest that graph theory teaching, research, and applications would benefit greatly from the use of a unified software environment for graph construction, interaction, and visualization.

Based on this need, a comprehensive survey was undertaken of existing graph theory software packages, programs, and libraries to determine the suitability of each for use as a graph theory teaching and research tool. Some of the desired components (especially in the realm of graph visualization) were found to be implemented in several current tools and systems, but no single system was located with the ability to perform all such functions together in a coordinated way.

Graph Shop (the Graph Theory Workshop) is a new software package for graph theory research and applications. It was designed to be usable by students and graph theory beginners yet powerful enough to assist with advanced graph theory research. It runs on a variety of platforms and is available for free under the GNU GPL open source license.

(49 pages)

ACKNOWLEDGMENTS

The author wishes to thank the following for their support, assistance, and contributions to this project:

Stefanie Andersen, David Brown, Myra Cook, Bjarne Stroustrup, Hank Turowski, Scott Fife, Janette Andersen, Ross McConnell, Daren Orth, U.S.R. Murty, Rilo Kiley, LeRoy Beasley, Christophe Paul, Bishop Wilson, Linus Torvalds, Haavard Nord, Jack Andersen, Stack Overflow, James Madison, Jan Marie Andersen, Kent Hansen, S. Truett Cathy, Little George, Cindy Moulton, Janice Wong, Google, Leonhard Euler, Laurent Viennot, Rodgers and Hammerstein, Doug Hunsaker, Minghui Jiang, Woodie Flowers, Michel Habib, Jan Andersen, Kimi Owens, Cyndi Rowland, Eirik Chambe-Eng, James Mullen, Ian Anderson, Dwight D. Eisenhower, Professor Cuddlebear, Sam Tarantino, Brendan Eich, David Andersen, Bishop Wilson, J.A Bondy, James Powell, Roy Allen

CONTENTS

Page

ABSTRACT....iii

ACKNOWLEDGMENTS....iv

LIST OF TABLES....vi

LIST OF FIGURES....vii

INTRODUCTION....1
- Graph Theory Terminology....1
- Overview of GraphShop....1
- Graph Theory Education, Research, and Applications....2
- Technical Specifications....3

GRAPHSHOP MAIN MENU....5
- File Menu....5
- View Menu....6
- Window Menu....6

GRAPH PANELS....7
- Adjacency Matrix Panel....8
- Drawing Panel....9
- Interval Graph Panel....14
- Tournament Panel....16

APPLICATION PANELS....18
- Script Editor Panel....18
- Code Editor Panel....18

GRAPH OPERATIONS....19
- Competition Graph....20
- Complement Graph....20
- Converse Graph....21
- Domination Graph....21
- Underlying Graph....22

SCRIPT API REFERENCE....23
- `window`....23
- `Graph`....24
- `Vertex`....31
- `Edge`....33
- `Arc`....34
- Derived Graphs....34

SCRIPT EXAMPLES....36

BIBLIOGRAPHY....39

APPENDIX....40

LIST OF TABLES

Table Page

1 Existing graph theory software.....41

LIST OF FIGURES

Figure Page

1 Graph Shop Interface.....1

2 Adjacency matrix panel.....9

3 Drawing panel.....10

4 Interval graph panel.....14

5 Tournament panel.....17

6 Graph Operations.....19

INTRODUCTION

Graph Theory Terminology

It is assumed that the reader is familiar with the foundational terms and concepts of mathematical graph theory. Detailed definitions and further information about basic and advanced graph theory topics can be found in the *Handbook of Graph Theory*.

In this text, the word graph is used to designate a general mixed multigraph (i.e., a graph in which any combination of edges, arcs, loops, and parallel edges/arcs are allowed). All GraphShop features have been designed to work with such general graphs except where specifically noted otherwise.

Overview of GraphShop

GraphShop is the Graph Theory Workshop, an interactive software environment for Graph Theory research and applications. It contains a visual "point-and-click" interface for graph creation and manipulations, as well as an integrated scripting environment (using ECMAScript) linked both ways with the GUI systems (Figure 1). This means that one can create a graph with scripting, examine and modify it with the visual interface, and then run a custom script algorithm on the result.

The design of GraphShop was motivated by an informed belief that a well-designed software

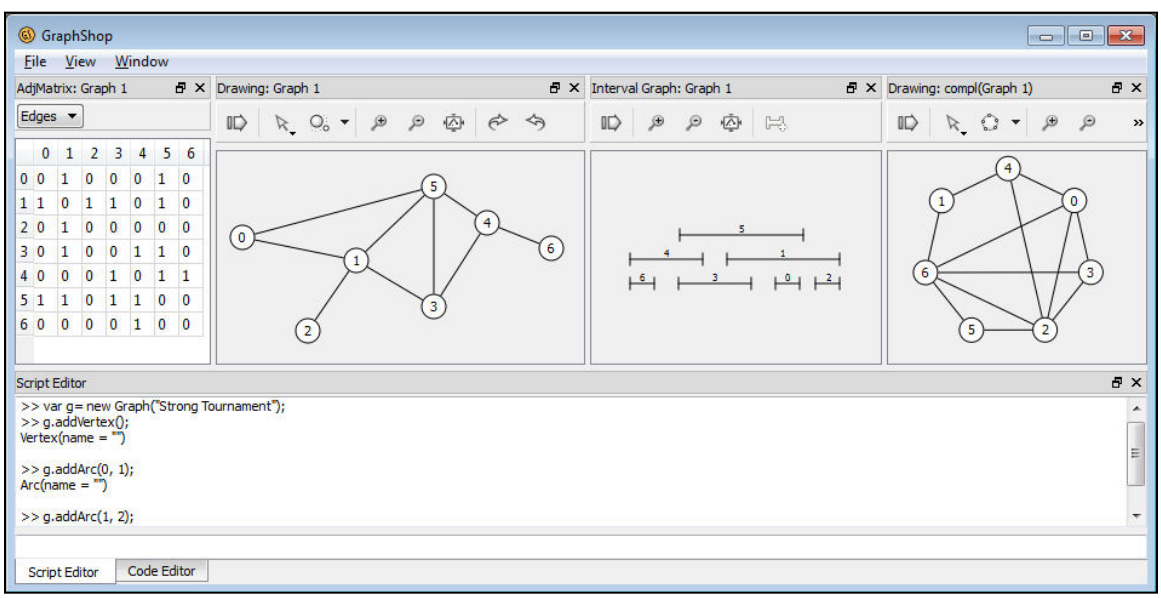


Figure 1. *Graph Shop Interface.*

system would be extremely valuable in graph theory research and applications and a determination that no current software package has the features and characteristics necessary to fill that role.

Graph Theory Education, Research, and Applications

It is hoped that GraphShop will be useful in three distinct but overlapping contexts: graph theory education, research, and applications.

In educational situations, the simple interface and visual components will assist students and beginners in quickly gaining an understanding of basic graph theory concepts and ideas. Professors and teachers of graph theory courses can use GraphShop as a tool for both demonstrating graph theory ideas and allowing their students to explore graph theory on their own. The ability to test and "try out" different types of graph modifications, operations, and constructions will help those students gain a mental model and intuition for more advanced graph theory concepts. Because GraphShop is open source and free, it can be widely distributed as necessary without imposing any additional financial burden on students or educational organizations.

Graph theory researchers will find GraphShop useful in a number of ways. It can quickly perform basic graph theory tasks and operations (such as simple graph generation and drawing) that are often difficult and time-intensive to do manually, thus freeing the researcher to focus more efficiently on the specific task at hand. The ability to save and load graphs to disk will assist collaboration by allowing multiple researchers to easily share and reference specific graphs or graph collections one with another. GraphShop's powerful scripting system allows for custom graph algorithms to be written and seamlessly used alongside its existing features and capabilities, allowing for efficient generation, modification, and classification systems to be created as necessary for individual groups and projects.

Those working in the realm of specific graph theory applications will appreciate that, although it contains powerful mathematical features and abilities, GraphShop does not assume advanced graph theory knowledge on the part of its users. Any person whose work intersects with graph theoretical structures should be able to use GraphShop for experimental or modeling purposes with only the basic graph understanding that comes from the original source domain. Advanced users will appreciate that GraphShop's scripting abilities allow for extensive customization of its systems and

for it to better fit any desired use or application.

Technical Specifications

GraphShop is written in the C++ language and is in its present form approximately 6500 non-comment lines of code. It was designed to run on multiple platforms and operating systems (though currently only Windows is officially supported).

Extensive use is made of the Qt C++ framework. The current GraphShop release is based on Qt 4.7.0 and was primarily developed using version 2.0.1 of the Qt Creator IDE.

Qt is used in GraphShop in three major ways:

First, Qt's cross-platform GUI module is used to create GraphShop's graphical user interface, particularly the customizable graph panels (based on the QMainWindow/QDockWidget system) and the Drawing, Tournament, and Interval Graph panels (all of which use the QGraphicsScene/QGraphicsItem system to power their primary display and interactivity functions).

Secondly, GraphShop's core graph classes (`Graph`, `Vertex`, `Edge`, `Arc`, etc.) rely heavily on Qt's Signal/Slot interobject communication mechanism to send and receive notifications of structural changes. These notifications are especially used by the derived graph classes and by the user interface, both of which need to update their own systems or displays whenever their source graph is modified.

Lastly, GraphShop's powerful scripting environment is based on the C++/QtScript bindings provided by Qt's QtScript module. This module is powered by JavaScriptCore, the ECMAScript interpreter from the WebKit open-source web browser layout engine.

The initial Windows binary release of GraphShop was compiled by Qt Creator 2.0.1 using the MinGW port of GCC 4.0.4. There are no significant requirements for specific system resources or external libraries, which means that GraphShop should run without trouble on any operational computer running a 32- or 64-bit version of the Windows operating system.

The source code and Windows binaries for the initial release of GraphShop can be found on the accompanying CD. The most recent version can be obtained from the author's website at http://stringoftheseus.com/projects/GraphShop/.

EXISTING SOFTWARE

Prior to the initial GraphShop development work, a full review of currently-existing, potentially-related software packages was performed. The goal of this review process was to determine the extent that each existing package satisfied the requirements and features identified as necessary or desirable for a graph theory research tool, as previously discussed.

The discovery methods consisted mostly of internet searches, using both general web search engines, as well as mathematical communities and software directories. A total of 36 distinct software programs, projects, packages, and libraries were identified. After the initial list was compiled, each item was researched to determine its purpose, license, and primary capabilities. For a list of such software, along with a summary of the features and properties of each, see Table 1 in the Appendix.

Of the 36 projects identified, by far the largest group (approximately 20 of them) was primarily or entirely concerned with drawing and visual layout of graphs, with no significant mathematical capabilities or components. Although such programs can be extremely useful and valuable for visual and graphical purposes, their lack of graph-theoretical underpinnings significantly limits their usefulness as research tools.

Another six of the identified programs were created as software libraries (as opposed to fully functional programs), requiring them to be imported into another software project to be used. Such libraries are sometimes extremely powerful, but as they can only be used by those with advanced software development experience, their helpfulness to the graph theory field is somewhat limited.

Despite being marketed as graph theory software, several of the existing programs were found to be entirely focused on network flow theory, the small subset of graph theory that is often most interesting to computer scientists. (It is often referred to in computer science literature as "graph theory," as if that were the whole of the field.) The limited focus of these packages prevents them from being of any major assistance in the broader field of mathematical graph theory.

Of the remaining programs (beyond those mentioned above), most were either stalled, failed, incomplete, or commercial software with extremely high license fees. Even taking the commercial software into account, however, no single program or package was identified with the set of features and capabilities generally needed for research and applications in graph theory.

GRAPHSHOP MAIN MENU

GraphShop's main menu provides functionality for modifying the behavior of the GraphShop window itself: loading and saving graphs and projects and opening and closing graph panels.

File Menu

The File menu contains the following commands:

New Graph

The new graph submenu contains commands for adding new graphs to the current project. Currently, only the creation of null graphs (i.e., graphs with no vertices, and therefore, no edges) is supported.

Open Graph/Project

The open graph/project command is used to load one or more graphs from disk and add them to the current project. Currently only graphs saved in the GSG and GSP (GraphShop Graph and GraphShop Project, respectively) formats are supported.

Save Graph

The save graph submenu allows any graph in the current project to be saved to disk. The only currently supported format is GSG (GraphShop Graph), GraphShop's native graph serialization format. Saving a graph saves only the mathematical object, not any of the visual presentation properties or other data from the graph panels.

Save Project

Saving the project saves all the graphs currently open in the window to disk using GSP (GraphShop Project), a variation of the GSG format used by GraphShop to save individual graphs.

Exit

Selecting exit will close the GraphShop window. Any unsaved graphs will be lost.

View Menu

The view menu contains the graph submenus, through which graph panels and graph operations can be accessed, followed by the toggle commands for the application panels. See the sections on graph panels, graph operations, and application panels for more information.

Window Menu

Full Screen

Brings the GraphShop window in and out of fullscreen mode.

GRAPH PANELS

At the fundamental level, a graph is an abstract mathematical object with no inherent physical or visual properties. However, the simple structural nature of a graph gives rise to many convenient graphical and mathematical representations. In GraphShop, visual inspection, modification, and interaction with graphs is primarily carried out via one of several included graph panels, each of which represents a different way of looking at or thinking about a particular graph.

In addition to the general graph panels (those that apply to any possible graph), GraphShop supports a number of specific panels that apply only to graphs with a particular type of structure or property. These panels show a graph in the way best suited for the structure in question, while allowing for easy modification of the graph within those structural limitations.

Multiple visual panels can be opened for one graph, providing a way to view and interact with the same underlying mathematical object in various ways. Also, instances of the same panel type can be opened for multiple graphs at the same time, providing a way to demonstrate the similarities or differences between those graphs in the particular visual or mathematical sense that is highlighted by the specific panel used.

Two types of interaction and modification are available from within graph panels. First, one may modify the details of the particular view being offered by the given panel. As these details are properties of the visual representation and not the actual graph, such changes will not be reflected in the graph object or in other panels. Additionally, one may use the graph panels to edit the structure or properties of the graph itself in a convenient visual way. Since these modifications change the underlying mathematical object, they will be immediately reflected in the script APIs and in all other panels opened for the same graph.

Most graph panels contain a panel toolbar through which the view or representation of that panel can be customized or altered. Although the majority of these functions are specific to the individual panels, several common buttons appear on many of all such toolbars. These include the Export Image button, through which the current state of the panel can be saved as a PDF document or PNG image and standard zoom controls for selecting a wider or narrower view of the panel's visual space.

To toggle the visibility of a particular graph panel, pick a graph from GraphShop's View menu, then select the desired panel from the panel list in the resulting graph submenu. It is not currently possible to open more than once instance of a particular panel for the same graph.

The following graph panels are currently supported:

Adjacency Matrix Panel

For a given graph G with n vertices, the *adjacency matrix* M of G is defined to be the n by n matrix such that entry (i,j) in M is equal to the number of arcs from vertex i to vertex j (or edges between vertices i and j) in G.

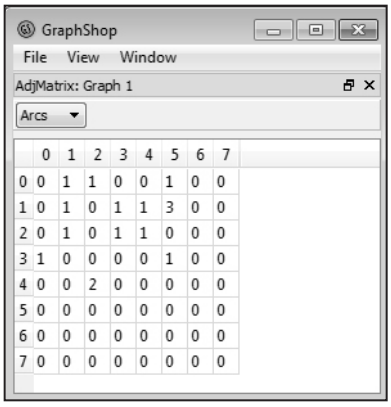
Because a graph can have both arcs and edges at the same time, GraphShop uses the term *arc adjacency matrix* for the first case and *edge adjacency matrix* for the second. Thus, every graph actually has two adjacency matrices (one for edges and one for arcs), though in the many cases (i.e., graphs that have only edges or only arcs) one or the other matrix will be empty.

The adjacency matrix highlights many important structural characteristics and properties of a graph and allows theorems and techniques from matrix theory and related mathematics disciplines to be applied to problems in graph theory. For this reason, many important proofs and discoveries in graph theory have risen out of adjacency matrix representations.

Additionally, it is worth noting that the adjacency matrix of a graph contains all of the structural information in the original graph, meaning that graphs can be completely described and represented by their adjacency matrices. In fact, the default format used by GraphShop to save a graph to disk is implemented via writing of the graph's adjacency matrix to a file.

GraphShop's adjacency matrix panel, as shown in Figure 2, presents a simple matrix view of the arcs or edges in the graph. The panel's toolbar contains a select list through which one may choose to view either the arc adjacency matrix or the edge adjacency matrix of the given graph.

To edit the adjacency matrix with the mouse, double click on the entry you wish to change, and type the new value in the resulting edit field. To quickly edit the matrix with the keyboard, navigate to the entry you wish to change (this can be done using the arrow keys), and begin typing the desired number. You can also use the tab key to quickly advance to and automatically begin editing the next entry in the matrix. Note that the edge adjacency matrix is automatically symmetric by definition, and



	0	1	2	3	4	5	6	7
0	0	1	1	0	0	1	0	0
1	0	1	0	1	1	3	0	0
2	0	1	0	1	1	0	0	0
3	1	0	0	0	0	1	0	0
4	0	0	2	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0

Figure 2. *Adjacency matrix panel.*

thus, will automatically update to reflect this as entries are added or changed.

Modifications to the adjacency matrix are automatically carried out in the given graph and immediately reflected in all other open panels.

There is not currently a way to add or remove vertices from a graph using the adjacency matrix panel.

Drawing Panel

The standard drawing of a graph—in which vertices are represented as circles or other simple shapes and edges and arcs are shown as lines and arrows, respectively—is so common and intuitive that it is often mistakenly thought of as being an intrinsic property of the graph itself. In reality, such a representation is just one of many different ways of visually presenting or "drawing" a particular graph. Determining the most efficient or aesthetically pleasing way to draw a given graph or creating a drawing with some desired property (such as a planar drawing in which none of the edge or arc lines cross) is such a large and interesting field that it has hundreds of published papers, several books, and its own annual conference.

In GraphShop, simple graph drawings can be created and manipulated using the graph drawing panel, seen in Figure 3. This panel allows for easy mouse-based positioning of the visual vertices in the graph, along with automatic positioning using one of several included algorithms.

The drawing panel toolbar contains a number of useful functions for manipulating or interacting with the displayed drawing or graph. In addition to the standard image export and zoom controls, the

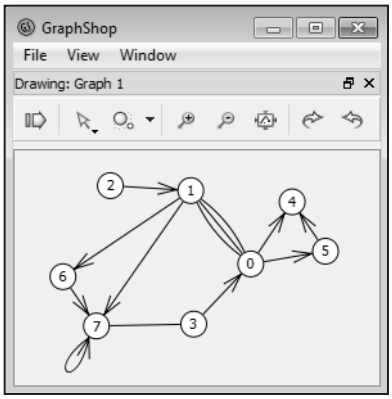


Figure 3. *Drawing panel.*

toolbar includes the following:

Interaction Mode

The interaction mode button is a list from which you can control how mouse clicks and movements should affect the drawing or graph structure. The following interaction modes are currently available:

• Select. This is the default mode which provides item selection and repositioning. Clicking on a drawing item (e.g., a vertex, edge, or arc) selects that item. Multiple items can be selected by holding down the Ctrl key while clicking. Starting in an empty location and dragging the mouse across the field creates a square "rubber band" box that selects everything within its bounds. Selected items can be removed from the graph by pressing the delete key, modified in a variety of ways using the panel's context menu, and have custom layout or rotations applied to them independent of the remainder of the graph.

Additionally (and perhaps most importantly), vertices in the drawing can be repositioned by dragging them within the visible drawing field with the mouse. Edges and arcs in the drawing are automatically updated to account for the new vertex positions. If multiple vertices are in a selection, those vertices can be dragged together, moving them to a new location while retaining their positions relative one to another.

• Pan. This mode allows for quick repositioning of the current view by automatically scrolling via a mouse drag. This is somewhat limited in its current form because it only allows for dragging within areas currently covered by the scrollbars.

• Add Vertex. In add vertex mode, clicking anywhere in the drawing field adds a new vertex to the graph and places its drawing in that location. This mode is a convenient way to add multiple new vertices to a graph in sequence and provides a way to establish the layout of the resulting drawing while doing so.

• Add Edge. To visually add a new edge to a graph, enable add edge mode, then click on two vertices in sequence. This will add a new edge between those two vertices. If either (or both) of those clicks is in an empty space in the drawing field (instead of on an existing vertex), a new vertex will be created in that location with the edge starting or ending from that vertex instead of an existing one. Double click on a vertex to create an edge loop or to finish the existing edge at that location and immediately start another edge from it.

• Add Arc. Similar to add edge mode, add arc mode allows you to add new arcs to the graph by clicking on existing vertices. A new arc can be created between one or more new vertices by clicking on empty spaces in the drawing field. Double click on a vertex to create an arc loop or to finish the existing arc at that location and immediately start another arc from it.

Layout

The graph drawing panel contains two methods of controlling the layout or position of the vertices, edges, and arcs being drawn. The first, custom drag-and-drop repositioning, is discussed in the section on the Select interaction mode. The second, the layout toolbar control, provides several automatic layout algorithms that can quickly position the vertices, edges, and arcs in the graph according to a particular predefined pattern.

The layout control is a combination button and menu. Use the arrow widget on the right side of the control to expose the list of layout algorithms, from which you can select an algorithm to run on the current graph. Use the button on the left side of the control to quickly rerun the currently selected layout algorithm at any time.

If two or more vertices (some algorithms also support application to a single selected vertex) in the drawing are currently selected, layout algorithms will only be applied to the selected vertices; Otherwise (if all vertices, no vertices, or for most algorithms, only one vertex is selected), layout algorithms will be applied to the entire graph. The ability to apply different automatic layouts to

different subsets of the drawing allows for quick customization.

The following layout algorithms are currently available:

• Random Layout. As its name implies, random layout is used to position the drawn vertices in a completely random way. When random layout is activated, the given vertices (the selected subset if there is one or more vertices selected, or the entire graph otherwise) are moved to random positions within the currently visible portion of the drawing field. If the drawing field is larger than the currently visible area (such that you need to use the scroll bars in order to see some of the vertices), random layout provides a quick way to bring all the vertices back into the currently visible area such that there are no longer scrollbars on the drawing field. The zoom controls can be used before running random layout to easily control the size of the area into which the vertices are randomly positioned.

• Circle Layout. Circle layout operates on two or more selected vertices or the whole graph and positions those vertices into a circular form relative to one another. The size of the circle is based on the number of vertices being positioned so as to create a consistent amount of space between the vertices regardless of the size of the vertex set involved.

Vertices in the circle are ordered according to their current position relative to their collective center. Before calculating new vertex positions, the circle layout algorithm first determines the center of the bounding rectangle of the existing vertex positions, then calculates the angle from the center of that rectangle to each vertex, and orders the vertices according to such angle. The effect of this is that you can control the sequence of the vertices around the resulting circle by first dragging them into a rough circle or arc in the order you want, then using circle layout to complete the process.

• Gravity layout. Gravity layout attempts to position two or more vertices in a natural or organic way using a simple energy minimization model. Laying out vertices using energy models is actually a fairly active research area within the graph drawing field, and many detailed and highly advanced algorithms have been published to do this in various efficient and interesting ways. GraphShop does not currently use any of those algorithms, mostly for reasons of implementation time.

Gravity layout functions by modifying the position of each vertex based on its relationship to the other vertices in the graph. For each pair of vertices, the algorithm first determines whether the two are neighbors, non-neighbors in the same component, or members of different components.

If two vertices are neighbors (i.e., there exists an edge or arc between them), they are considered to have a particular "ideal" distance apart, such that vertices closer than that will be moved slightly farther away from each other, and vertices farther away than that will be moved slightly closer together. Likewise, vertices in different connected components (i.e., vertices that have no set of edges or arcs connecting them) are positioned according to a similar (though numerically different) ideal separation difference. The remaining vertices (i.e., those that are in the same connected component, but not direct neighbors) are considered to have an ideal separation distance of infinity, meaning that the algorithm attempts to position them as far away as possible.

After a vertex has been compared to every other vertex in the graph, the different forces are combined into a single force vector, and the vertex is moved slightly in the direction of that vector. When all vertices have been so moved, the algorithm starts over with the calculations again and continues doing so (with a small delay between rounds) until no vertex has moved more than a certain threshold. Visually updating the drawing field after each round doesn't affect the final vertex positions but provides for an amusing (and somewhat educational) animation in which the vertices in the graph can be seen to slowly move around into an equilibrium state relative to each other. The name "gravity layout" is inspired by the mental image that comes from watching the process of the vertices moving as celestial bodies or subatomic particles.

The current gravity layout works well for most sparse and medium-density graphs but generally seems to produce somewhat crowded final positions when applied to graphs with a large number of edges and arcs relative to the size of the vertex set. This could be fixed with a modified algorithm, perhaps (as previously mentioned) using an existing published algorithm designed and tested by someone else.

Rotation

The rotation controls available in the drawing panel function very much like the automatic layout algorithms discussed previously. Similar to the graph layouts, graph rotations operate on two or more selected vertices, or the entire graph if less than two vertices are selected. The clockwise and counterclockwise rotations occur relative to the center of the bounding rectangle of the vertices being rotated, not the center of the drawing field or visible area. This generally provides for a

smoother visual experience, especially when rotating a subset of vertices with a larger graph.

Interval Graph Panel

An interval graph is a graph that can be represented as a sequence of open or closed intervals on the real line. In such a representation, each interval represents a vertex in the graph with an edge between two given vertices if and only if their corresponding intervals have a non-empty intersection. Not every graph has an interval representation, but those that do (i.e., the interval graphs) share enough common properties and real-life applications to make the study of interval graphs a vibrant research area within the field of graph theory.

GraphShop's interval graph panel, shown in Figure 4, provides a way to determine whether or not a given graph has an interval representation, and (in the case that it does) to view and interact with one such representation. Intervals can be moved horizontally by dragging their centers with the mouse or resized to the left or right by dragging their endpoints. To simplify the visual layout, intervals are automatically stacked vertically so that intersecting intervals do not have overlapping drawings. (This vertical order is purely stylistic and has no meaning in the graph or interval graph models.)

If the source graph is found to not be an interval graph (i.e., there is no possible interval representation of it), the interval graph panel displays the message "invalid interval graph." The graph will be reanalyzed after any future structural modifications, such that should it later become a valid interval graph the panel will immediately be updated to reflect that situation.

The interval graph panel toolbar contains an "add interval" button through which a new interval

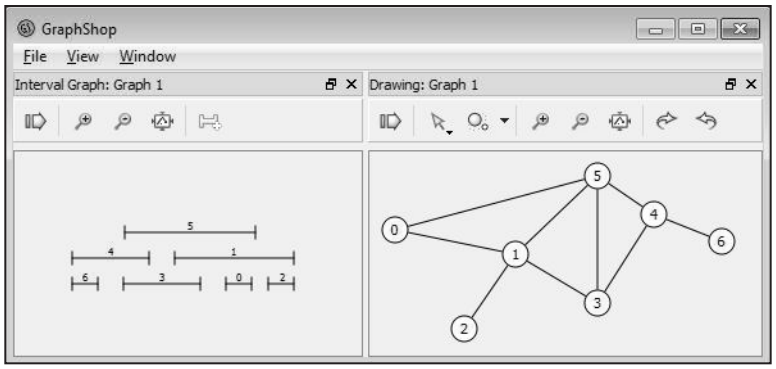


Figure 4. *Interval graph panel.*

can be added to the interval graph. Currently this interval is always added to the empty area to the right of the existing intervals.

Updating the core graph object to reflect changes made to the interval graph is a fairly simple process—one just has to make sure there is always one vertex per interval and watch for simple interval intersections. Going the other direction—taking an arbitrary graph and attempting to find an interval representation of it or determining whether or not such a representation even exists—is significantly more complicated. Luckily, graph theorists have been working on that problem for quite a while, and have come up with various ways of doing it. During the original development of the interval graph panel, several published algorithms were located and analyzed for their simplicity, flexibility, efficiency, and general suitability for use as part of GraphShop, with initial prototype implementations attempted for the top contenders.

Actually implementing the published algorithms proved to be a much more difficult task than originally anticipated. A friend of the author once joked that in order to successfully publish a paper in mathematics, one must first write a fifteen-page proof, then remove a random subset of four consecutive pages and replace their contents with the phrase "it is clear that…" While that may not be true of mathematics research in general, several of the algorithm papers encountered as part of this process seemed to be based on that philosophy. Phrases such as "one can then easily verify that…" or "with efficient data structures, … can be accomplished in linear time" appeared frequently, with no further explanation or hints as to what the implementer should do or where he should go for more information. Additionally, it was common for published papers to not actually contain the entire algorithm but to cite several other papers for necessary parts; in many cases, the referenced papers themselves required additional papers, with each successive round older and further from the original problem than the last. Although this web of references would make for an interesting directed graph, it makes producing a working implementation of the published algorithm a near impossible task; it is entirely possible that even the originals' authors themselves never had a working version but only theoretically proved their algorithm successful by relying on the written conclusions in previously published work.

The recognition component of the current interval graph panel is based on an interval graph recognition algorithm by Michel Habib et al. (from a 2000 paper titled "Lex-BFS and partition

refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing"). This algorithm was selected because it is fairly simple to understand, is largely free from the common problems mentioned above, and uses data structures that are easily adapted to existing GraphShop systems and APIs. The GraphShop implementation has three major differences from the one in the paper. First, it uses a standard nlogn sorting function instead of the two-pass radix sort suggested, which means that it does not quite achieve the same running time as the published version. (This was done primarily for reasons of implementation time.) Second, since the published algorithm implicitly assumes that the input graph is connected (which is quite often not the case in real life), GraphShop first splits the graph into its individual connected components, runs the recognition algorithm on each component, and finally combines the resulting clique chains into a single master chain before drawing the associated intervals. Lastly, the GraphShop algorithm corrects several small errors in the paper's original pseudocode which would have rendered an exact implementation nonfunctional.

Tournament Panel

A tournament is a complete simple digraph (i.e., a graph in which there is exactly one arc between every pair of distinct vertices). The name tournament comes from using these graphs as models for sporting events and other competitions in which every player or team competes once against every other, and every competition has a clear winner. Because of this association, and because of a host of interesting mathematical and structural properties shared by all such graphs (along with an even larger set of properties surprisingly not consistent across tournaments, despite their constrained structure), tournaments have been and are well studied and used in graph theory research and applications.

Because tournaments are by definition extremely dense and "similar-looking" graphs, the standard "circles and arrows" graph drawing method is often not very useful, especially when trying to quickly determine the differences or similarities among multiple tournaments. For this reason, tournaments are commonly visualized using a different method in which the vertices of the graph are laid out in a horizontal or vertical line, with the arcs that would be directed down or to the left omitted from the drawing. Splitting the arcs into groups of "what's there" and "what's not" makes it

easy to see which direction any particular arc is going and often highlights interesting properties of the tournament being drawn.

GraphShop's tournament panel (Figure 5) provides a way to determine if the source graph is or is not a valid tournament and to draw it in standard tournament form if so. Vertices in the tournament drawing are arranged horizontally and can be reordered by dragging them to different positions. Rather than being omitted, arcs directed to the left are drawn in a light but visible grey so as to facilitate their selection with the mouse (after which their direction can be flipped using the panel's context menu).

In addition to the standard panel controls, the tournament panel toolbar contains a button to add a vertex to the tournament. Doing so automatically adds an arc between the new vertex and all existing vertices so as to maintain the tournament properties of the graph.

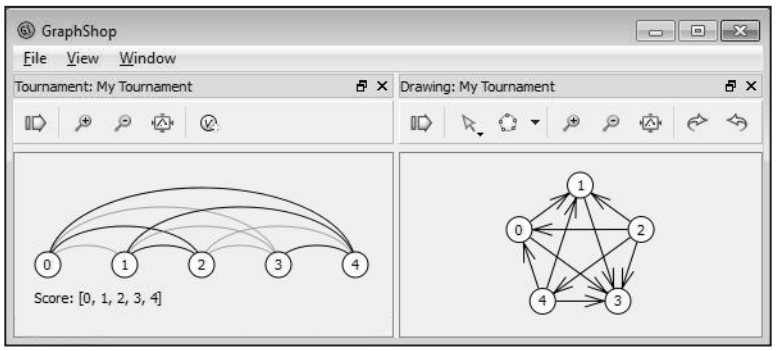


Figure 5. *Tournament panel.*

APPLICATION PANELS

GraphShop's application panels operate similarly to the various graph panels, only they provide features and functionality related to the entire GraphShop application (as opposed to one specific graph). The currently implemented application panels are listed at the bottom of the View menu below the individual graph menus. Once opened, application panels can be toggled, moved, resized, and intermingled with visible graph panels in any way desired.

Script Editor Panel

The script editor panel provides a simple command-line interface to the GraphShop scripting API through which a single function, action, or command can be immediately executed. To use the script editor panel, type your desired script one line at a time into the input box near the bottom of the panel, then press enter to execute that line of code. (In some cases, such as when the current code sequence is part of a loop or other incomplete control structure, the actual execution may be delayed until the completion of that structure is completed.)

The script editor panel is ideal for quick execution of single script commands or short functions. For longer scripts and repeated operations, consider using the code editor panel instead.

For details of the functions and classes available in the scripting environment, see the Script API Reference. For more information about ECMAScript and learning resources, see the Bibliography.

Code Editor Panel

To create advanced graph scripts or reusable script functions, use GraphShop's code editor panel. The script editor panel consists of a large text input box into which a graph script or function can be composed or entered. Currently, this is a plain text input field without any of the more advanced features common to many code editors and software development environments.

The script and code editor panels share a single backend engine and global scope. This means that any variables assigned in the code editor can be accessed from the script editor and vice versa. This is especially useful in the case of functions, where a named function can be defined in the code panel, then quickly called or referenced from the script panel.

For details of the functions and classes available in the scripting environment, see the Script API Reference. For more information about ECMAScript and learning resources, see the Bibliography.

GRAPH OPERATIONS

GraphShop supports a number of unary graph operations through which a new graph is defined based on the structure or properties of an original. These graphs (referred to as derived or dependent graphs, see Figure 6) can be created automatically for any available graph, including other derived graphs. To toggle panels corresponding to a derived graph, find the graph submenu for the source graph in GraphShop's View menu, then select the desired operation from directly beneath the list of graph panels. The resulting menu will be the graph menu for the implicitly created derived graph, through which one may open any available panel for that graph or create further derived graphs as deep as desired.

Derived graphs are dynamically linked to their source graphs and updated automatically when necessary. Thus, any change to a graph (via a panel or scripting) will automatically be reflected in any corresponding derived graphs (and the associated graph panels).

In scripting, live derived graphs may be created by passing the original graph as a construction parameter to any of the derived graph classes. Because the initial calculation of derived graphs can sometimes be computationally intensive (especially for large graphs), these calculations are not performed immediately upon creation of a derived graph object; invoke the `build()` method on the derived graph object before using it or any of its inspection functions.

Static derived graphs (i.e., derived graphs that are not affected by future modifications to the

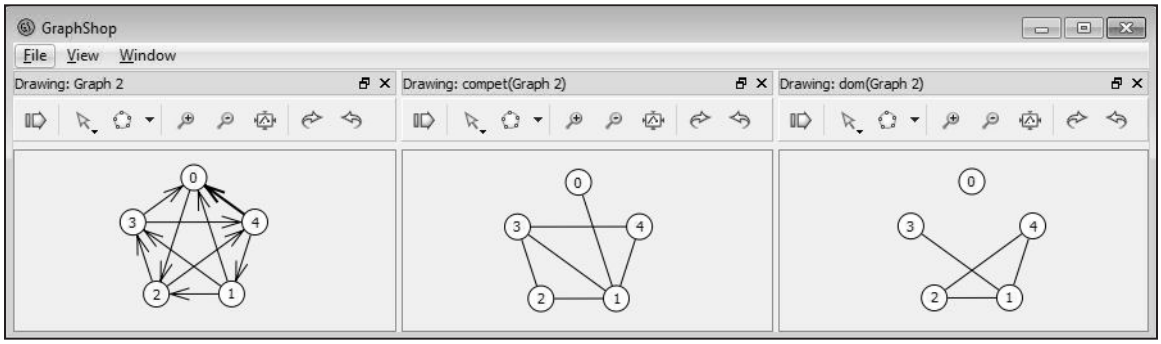


Figure 6. *Graph Operations*.

original graph) can be obtained in scripting by calling one of the getXX (where XX is the name of one of the derived graphs described below) derived graph methods on the original graph object. Because they depend on the exact current state of the source graph, static derived graphs are built automatically when created.

Although the mathematical definitions may sometimes be extended to include arbitrary mixed graphs, the graph operation algorithms as implemented in GraphShop usually assume a simple graph or digraph as input. The behavior and output of these algorithms when performed on mixed or complex graphs is unpredictable and not guaranteed.

The following operations are currently supported.

Competition Graph

For a given digraph G=(V, A), two vertices x, y in V are said to compete in G if there exists a vertex z in V such that x beats z and y beats z (i.e., arcs (x, z) and (y, z) exist in G). The *competition graph* compet(G) =(V, E) is an undirected graph in which edge (x, y) exists in E if and only if vertices x and y compete in the original graph G.

The concept of a competition graph originally arose via modeling of ecological systems as directed graphs. Specifically, if the vertices in a digraph represent various plant and animal species in a particular environment, with every arc (t, h) signifying that species t eats species h, then two species x and y which both eat the same species z can be said to "compete for" z. The competition graph of the complete "who eats what" digraph models all the competition relationships in the original system.

In scripting, the live competition graph of a given graph g can be created using `new CompetitionGraph(g)`, where the new graph's `build()` method will need to be invoked before it can be used; alternately, an automatically-built static competition graph can be obtained using `g.getCompetitionGraph()`.

Complement Graph

For a given undirected graph G=(V, E), the *complement graph* (also called the inverse graph) compl(G) = (V, E') is an undirected graph such that edge {x, y} exists in E' if and only if the same

edge does not exist in E. It is clear that if C=(V,U) is the complete graph on V (i.e., U is the set of all possible edges between the vertices in V), then E' is the set complement of E in U. Additionally, the union of G and compl(G) is the complete graph C, which allows for the definition that compl(G) = C – G.

In scripting, the live complement graph of a given graph g can be created using `new ComplementGraph(g)`, where the new graph's `build()` method will need to be invoked before it can be used; alternately, an automatically-built static complement graph can be obtained using `g.getComplementGraph()`.

Converse Graph

For a given digraph G=(V, A), the *converse graph* (also called the transpose or reverse graph) conv(G) =(V, B) is a digraph such that arc (x, y) exists in B if and only if arc (y, x) exists in A. Thus, the converse graph is the graph obtained by reversing the direction of all the arcs in a particular digraph.

The term *transpose graph* is a reference to the fact that the converse graph is the graph represented by the transpose of the directed adjacency matrix of the original graph.

In scripting, the live converse graph of a given graph g can be created using `new ConverseGraph(g)`, where the new graph's `build()` method will need to be invoked before it can be used; alternately, an automatically-built static converse graph can be obtained using `g.getConverseGraph()`.

Domination Graph

For a given digraph G=(V, A), a subset S of V is said to be a *dominating set* of G if for every vertex v in V, either v is in S or there exists some vertex s in S such that the arc (s, v) is in A. The *domination graph* dom(G)=(V, B)) is an undirected graph such that for every pair x, y in A, edge {x, y} exists in B if and only if the set {x, y} forms a dominating set of G.

In scripting, the live domination graph of a given graph g can be created using `new DominationGraph(g)`, where the new graph's `build()` method will need to be invoked before it can be used; alternately, an automatically-built static domination graph can be obtained using

`g.getDominationGraph()`.

Underlying Graph

For a given digraph G=(V, A), the *underlying graph* under(G) = (V, E) is an undirected graph such that edge {x, y} is in E if and only if either of arcs (x, y) or (y, x) exist in A. Conceptually, the underlying graph of a digraph is created by replacing every arc in the original digraph with an edge between the same two vertices.

In scripting, the live underlying graph of a given graph g can be created using `new UnderlyingGraph(g)`, where the new graph's `build()` method will need to be invoked before it can be used; alternately, an automatically-built static underlying graph can be obtained using `g.getUnderlyingGraph()`.

SCRIPT API REFERENCE

The GraphShop scripting environment has access to all the standard global variables, functions, and objects from the ECMAScript specification, such as `Math`, `Array`, `parseInt()`, etc. This corresponds to the typical global scope found in JavaScript, with the exception of the web-specific items such as the document object and the DOM API.

For more information on these build-in objects, as well as the ECMAScript language itself, consult any good book or website on JavaScript (ECAMAScript's most widely used implementation). Some suggested ECMAScript resources can be found in the Bibliography.

`window`

The global variable window is available in all script contexts and provides a reference to GraphShop's graphical user interface. The window object has several scriptable methods, which are primarily used to work with the list of graphs currently available in the ui.

`Array<Graph> graphList()`

The list of graphs currently available in the user interface.

`Graph getGraph(int)`

Get a reference to a graph from the user interface by index number. Index numbers start at 0 and increment for each new graph added to the ui, and correspond to the order in which the graphs are listed in GraphShop's View menu.

`int graphCount()`

The total number of graphs currently available in the user interface.

`Graph addGraph(Graph)`

Add an existing graph from the scripting environment to the user interface.

`Graph includeGraph(Graph)`

Verify that the given graph is part of the window graph list. Unlike `addGraph`, this function does nothing if the given graph has already been added.

`Graph addNewGraph(String label="")`

Create a new graph and immediately add it to the window graph list. A reference to the newly created graph is returned.

`Graph`

To construct a new graph with a given label, use `new Graph(label)`. The label parameter can be omitted, which will create a graph with an empty label. The graph will be initially null (i.e., with no vertices, edges, or arcs).

`Graph clone()`

Create and return a new graph identical in structure to this one. Graph and vertex labels will be copied from the original.

`void build()`

For graphs created using the graph constructor, this does nothing. Derived graphs (i.e., graphs created using one of the graph operation constructors) need to be built using `build()` before they can be used.

`Vertex addVertex(String vertexLabel = "")`

Creates a new vertex with the given label (or an empty label if the label parameter is omitted) and adds it to the graph. The newly created vertex is returned so that it can be stored or used if necessary.

`void removeVertex(Vertex vertex)`

Remove the given vertex from the graph. Any edges or arcs connected to that vertex will be removed from the graph also. The removed objects should then be considered invalid, and any further references to them held by the calling script discarded.

`void removeVertices()`

Remove all vertices from the graph. This has the effect of clearing out the all the edges and arcs also, resulting in a null or empty graph.

`Vertex getVertex(int index)`

Returns the vertex with the given index, where vertices in an n-vertex graph are numbered 0 through n-1. The order of the vertices in this system has no defined meaning and is only guaranteed to be constant for a given vertex as long as no other vertices are added or removed from the graph. The vertex index is thus not useful as a long-term identifier but is extremely valuable for iterating over the vertices in a graph and for short-term references within a particular algorithm or script.

`Array<Vertex> getVertexSet()`

Returns an array containing all the vertices in the graph. For simple iteration over the vertex set it is generally better to use `getVertex` with `vertexCount` instead.

`int vertexCount()`

The number of vertices in the graph, otherwise known as the order of the graph.

`Edge addEdge(int vertex1, int vertex2, String edgeLabel = "")`

Adds a new edge to the graph, with an optional label, using the indices of the endpoint vertices instead of the vertices themselves. A reference to the newly added edge is returned.

`Edge addEdge(Vertex vertex1, Vertex vertex2, String edgeLabel = "")`

Adds a new edge to the graph, with an optional label, using a reference to each endpoint vertex. A reference to the newly added edge is returned.

`void removeEdge(Edge)`

Remove the given edge from the graph. To remove an edge by index, call `removeEdge(getEdge(index))`.

`void removeEdges()`

Remove all the edges in the graph. If the graph contains no arcs, this results in a graph consisting entirely of isolated vertices.

`int edgeMultiplicity(int vertex1, int vertex2)`

The number of edges between the given vertices, referenced by index.

`int edgeMultiplicity(Vertex vertex1, Vertex vertex2)`

The number of edges between the given vertices. This is here as a shortcut for

`getEdges(vertex1, vertex2).length` and for consistency with `setEdgeMultiplicity`.

`void setEdgeMultiplicity(int vertex1, int vertex2, int count)`

Let there be count edges between the given vertices, referenced by index.

`void setEdgeMultiplicity(Vertex vertex1, Vertex vertex2, int count)`

Let there be count edges between the given vertices. This could require adding or deleting edges from the graph depending on how many such edges there were before. If new edges are added, they will be given blank labels.

`Edge getEdge(int index)`

Get the edge with the given index. Index values range from 0 to `edgeCount()-1` and are generally the most convenient way of identifying a particular edge within an algorithm. The index of a given edge has no defined meaning and is likely to change if edges are added to or removed from the graph.

`bool hasEdge(int vertex1, int vertex2)`

True if there is at least one edge between the given vertices, identified by index.

`bool hasEdge(Vertex vertex1, Vertex vertex2)`

True if there is at least one edge between the given vertices. This is a shortcut for `getEdges(vertex1, vertex2).length > 0`, though it is implemented in a way that is likely to be faster than that.

`Array<Edge> getEdges(int vertex1, int vertex2)`

Returns an array containing all the edges between the given vertices, identified by index.

`Array<Edge> getEdges(Vertex vertex1, Vertex vertex2)`

Returns an array containing all the edges between the given vertices. If no such edges exist, the result is an empty array.

`Array<Edge> getEdgeSet()`

Get an array containing all the edges in the graph. For simple iteration over the `edgeSet`, it is generally better to use `getEdge` with `edgeCount` instead.

`int edgeCount()`

The number of edges in the graph, otherwise known as the size of the graph. If the graph contains both edges and arcs, the actual graph size is `edgeCount() + arcCount()`.

`Arc addArc(int tail, int head, String arcLabel = "")`

Add a new arc, with an optional label, from vertex tail to vertex head, identified by their indices. A reference to the newly created arc is returned.

`Arc addArc(Vertex tail, Vertex head, String arcLabel = "")`

Add a new arc, with an optional label, from vertex tail to vertex head. A reference to the newly created arc is returned.

`void removeArc(Arc)`

Remove the given arc from the graph. To remove an arc by index, call `removeArc(getArc(index))`.

`void removeArcs()`

Remove all arcs from the graph. If the graph contained only arcs (i.e., no edges), this results in a graph consisting of isolated vertices.

`int arcMultiplicity(int tail, int head)`

The number of arcs with the given head and tail vertex, identified by index..

`int arcMultiplicity(Vertex tail, Vertex head)`

The number of arcs with the given head and tail vertex. This is here as a shortcut for `getEdges(vertex1, vertex2).length` and for consistency with `setEdgeMultiplicity`.

`void setArcMultiplicity(int tail, int head, int count)`

Let there be count arcs going from vertex tail to vertex head, identified by index.

`void setArcMultiplicity(Vertex tail, Vertex head, int count)`

Let there be count arcs going from vertex tail to vertex head. This may require adding or removing arcs from the graph, depending on its current structure. If new arcs are added, they will have empty labels. Arcs going the opposite direction (i.e., from vertex head to vertex tail) are not affected.

`Arc getArc(int index)`

Get the arc with the given index. Index values range from 0 to `arcCount()-1` and are generally the most convenient way of identifying a particular arc within an algorithm. The index of a given arc has no defined meaning and is likely to change if arcs are added to or removed from the graph.

`void flipArc(Arc arc)`

Flip the direction of the given arc (i.e., set the arc to go from its current head to its current tail, so that the current tail becomes the new head and vice versa).

`bool hasArc(int trail, int head)`

True if there exists at least one arc from vertex tail to vertex head, identified by index.

`bool hasArc(Vertex tail, Vertex head)`

True if there exists at least one arc from vertex tail to vertex head. This is a functionally equivalent to `getArcs(tail, head) > 0` but is likely to be slightly faster especially in large graphs.

`Array<Arc> getArcs(int tail, int head)`

Returns an array containing all the arcs going from tail to head, identified by index.

`Array<Arc> getArcs(Vertex tail, Vertex head)`

Returns an array containing all the arcs from tail to head. If no such arcs exist, an empty array is returned.

`Array<Arc> getArcSet()`

Returns an array containing all the arcs in the graph. For simple iteration over the arc set, it is generally better to use `getArc` with `arcCount` instead.

`int arcCount()`

The number of arcs in the graph. In the context of a pure digraph, this is often called the size of the graph (though if the graph also contains edges, the true size is `arcCount() + edgeCount()`).

`void clear()`

Removes all the edges, arcs, and vertices from the graph. This is functionally equivalent to `removeVertices()` but is implemented as a single step instead of a series of individual item deletions.

`Array<Array<Vertex> > getComponents()`

Calculate and return the connected components (often just called the components) of the graph. The return value is an array of arrays of vertices, where each top-level array represents a component and each component is represented by its member vertices.

`String label()`

The label of the graph. Graph's created from the user interface are given default labels of the form "Graph n," where n is the window index number of the new graph. Graphs created from scripting default to an empty label.

`void setLabel(String newLabel)`

Set the graph label to the given string. It is recommended that graph labels be unique to the extent possible, though no system relies on this property so it is not required.

`bool isComplete()`

True if the graph is either arc complete or edge complete. This fact is rarely useful, but serves as a shortcut for calling `isArcComplete()` or `isEdgeComplete()` in situations when the graph is known to be pure (i.e., containing only edges or only arcs).

`bool isArcComplete()`

True if there is at least one arc between every two vertices. This is still true if there is more than one such arc; if it is desired to know whether the graph contains one and only one arc between every two vertices, further testing to verify that `arcCount() = (n*(n-1))/2` where `n = vertexCount()` would be required.

`bool isEdgeComplete()`

True if there is at least one edge between every two vertices. This is still true if there is more than one such edge; if it is desired to know whether the graph contains one and only one edge

between every two vertices, further testing to verify that `edgeCount() = (n*(n-1))/2` where `n = vertexCount()` would be required.

`bool isCompleteGraph()`

True if the graph is both a pure graph (i.e., has no arcs) and edge complete.

`bool isCompleteDigraph()`

True if the graph is both a pure digraph (i.e., has no edges) and arc complete.

`bool isPure()`

True if the graph is either a pure graph or a pure digraph.

`bool isPureGraph()`

True if the graph has no arcs (presumably, this means it has edges, though the null graph also qualifies).

`bool isPureDigraph()`

True if the graph has no edges (presumably, this means it has arcs, though the null graph also qualifies).

`bool isEmpty()`

True if the graph has neither edges nor arcs. An empty graph is the collection of zero or more isolated vertices.

`bool isNull()`

True if the graph has no vertices (and thus no edges or arcs).

`bool isTrivial()`

True if the graph contains exactly one vertex (and no edges or arc loops). This is often more appropriately called the singleton graph.

`Graph getCompetitionGraph()`

Returns a static copy of the competition graph of this graph. To create a linked competition graph (i.e., a graph that automatically updates to reflect changes to the structure of this one), use the `CompetitionGraph` constructor.

See the derived graphs section of the user interface documentation for the definition and

explanation of the Competition Graph.

`Graph getComplementGraph()`

Returns a static copy of the complement graph of this graph. To create a linked complement graph (i.e., a graph that automatically updates to reflect changes to the structure of this one), use the `ComplementGraph` constructor.

See the derived graphs section of the user interface documentation for the definition and explanation of the Complement Graph.

`Graph getConverseGraph()`

Returns a static copy of the converse graph of this graph. To create a linked converse graph (i.e., a graph that automatically updates to reflect changes to the structure of this one), use the `ConverseGraph` constructor.

See the derived graphs section of the user interface documentation for the definition and explanation of the Converse Graph.

`Graph getDominationGraph()`

Returns a static copy of the domination graph of this graph. To create a linked domination graph (i.e., a graph that automatically updates to reflect changes to the structure of this one), use the `DominationGraph` constructor.

See the derived graphs section of the user interface documentation for the definition and explanation of the Domination Graph.

`Graph getUnderlyingGraph()`

Returns a static copy of the underlying graph of this graph. To create a linked underlying graph (i.e., a graph that automatically updates to reflect changes to the structure of this one), use the `UnderlyingGraph` constructor.

See the derived graphs section of the user interface documentation for the definition and explanation of the Underlying Graph.

`Vertex`

To create a new vertex, call `addVertex` on the graph it should belong to. Vertices cannot be

created outside of a graph, nor can a vertex be moved from one graph to another.

`Array<Vertex> neighborhood()`

Returns an array with references to all the vertices that are neighbors of this one (i.e., all the vertices that are connected to this one by an edge or arc going either direction). If no such vertices exist (i.e., if this is an isolated vertex), the resulting array will be empty.

`Array<Vertex> inNeighborhood()`

Returns an array with references to all the vertices in this vertex's in neighborhood (i.e., all the vertices for which there is an arc going from that vertex to this one). If no such vertex exists, the resulting array will be empty.

`Array<Vertex> outNeighborhood()`

Returns an array with references to all the vertices in this vertex's out neighborhood (i.e., all the vertices for which there is an arc going from this vertex to that one). If no such vertex exists, the resulting array will be empty.

`int index()`

The current index of this vertex in the parent graph. This value can change as vertices are added to or deleted from the graph.

`String label()`

The vertex label, as given to the parent graph's `addVertex` function or set with `setLabel`.

`void setLabel(String newLabel)`

Sets the vertex label to the given string.

`Array<Edge> edges()`

All the edges connected to this vertex. To add or remove edges from the vertex, use the appropriate edge function on the parent graph object.

`Array<Edge> edgesWith(Vertex other)`

All the edges between this vertex and the given other. If no such edges exist, the resulting array will be empty.

`Array<Arc> arcs()`

All the arcs connected to this vertex (going either direction).

`Array<Arc> arcsWith(Vertex other)`

All the arcs connected to this vertex and the given other (going either direction).

`Array<Arc> inArcs()`

All the arcs coming into this one (i.e., the arcs for which this vertex is the head).

`Array<Arc> arcsFrom(Vertex other)`

All the arcs coming from the given other vertex to this one (i.e. , with the other vertex and this one as their tail and head, respectively).

`Array<Arc> outArcs()`

All the arcs going out from this one (i.e., the arcs for which this vertex is the tail).

`Array<Arc> arcsTo(Vertex other)`

All the arcs going from this vertex to the given other (i.e., with this vertex and the other as their tail and head, respectively).

`Edge`

To create a new edge, call `addEdge` on the graph it should belong to. Edges cannot be created outside of a graph, nor can an edge be moved from one graph to another.

`Vertex vertex1()`

The vertex connected to the "first" end of this edge.

`Vertex vertex2()`

The vertex connected to the "second" end of this edge. In reality, the first and second ends of an edge are an unordered pair; the distinction between `vertex1` and `vertex2` is created solely to allow them to be referenced separately in scripting. Although this distinction implies a sort of direction to an edge, that property is meaningless and should generally not be used as part of any algorithm or script.

`int index()`

The current index of this edge in the parent graph. This value can change as vertices are added to or deleted from the graph.

`String label()`

The edge label, as given to the parent graph's `addEdge` function or set with `setLabel`.

`void setLabel(String newLabel)`

Set the edge label to the given string.

`Arc`

To create a new arc, call `addArc` on the graph it should belong to. Arcs cannot be created outside of a graph, nor can an arc be moved from one graph to another.

`Vertex head()`

The head or ending vertex of this arc.

`Vertex tail()`

The tail or starting vertex of this arc. An arc is said to go from its tail to its edge which is implied by stating that the tail vertex beats the head vertex.

`int index()`

The current index of this arc in the parent graph. This value can change as vertices are added to or deleted from the graph.

`String label()`

The arc label, as given to the parent graph's `addArc` function or set with `setLabel`.

`void setLabel(String newLabel)`

Set the arc label to the given string.

Derived Graphs

Derived graphs (i.e., graphs created from the various graph operations) can be obtained from

an existing graph (including another derived graph) by constructing a new derived graph object (i.e., a new object using the `CompetitionGraph`, `ComplementGraph`, `ConverseGraph`, `DominationGraph`, or `UnderlyingGraph` constructors) and passing in the source graph as the sole construction parameter.

Derived graphs have the same features and API as the regular `Graph` class. Once created, derived graphs are automatically updated to reflect changes made to the structure of the source graph. For this reason, the graph modification function should not be called directly on a derived graph. To get a static copy of a derived graph at any time, use the `clone()` function.

SCRIPT EXAMPLES

The following examples demonstrate how easy it is to write custom graph functions and algorithms using GraphShop's scripting module. To use one of these functions in your own scripts, copy and paste the code into the code editor window. You may then write the remainder of your script below the function definition, or immediately click "run" to parse and define the function. The function can then be referenced and called from either the code editor or the script editor windows. Once defined, a function's source may (if desired) be removed from the code editor window.

Complete graph on n vertices

```
function completeGraph(n)
{
	var g = new Graph("K"+n);
	for(var i=0; i<n; i++)
	{
		for(var j=i+1; j<n; j++)
		{
			g.addEdge(i, j);
		}
	}

	return g;
}
```

Random n-tournament

```
function randomTournament(n)
{
	var g = new Graph("T"+n);

	for(var i=0; i<n; i++)
	{
		for(j=i+1; j<n; j++)
		{
			if(Math.random() > 0.5)
			{
				g.addArc(i, j);
			}
			else
			{
				g.addArc(j, i);
			}
		}
	}

	return g;
}
```

Random n-graph with edge probability p

```
function randomGraph(n, p)
{
	var g = new Graph("R"+n);

	for(var i=0; i<n; i++)
	{
		for(j=i+1; j<n; j++)
		{
			if(Math.random() < p)
			{
				g.addEdge(i, j);
			}
		}
	}

	return g;
}
```

Simple edge-union

```
function graphUnion(g1, g2)
{
	var n1 = g1.vertexCount();
	var n2 = g2.vertexCount();
	var ng = Math.max(g1, g2);

	var g = new Graph(g1.label() + " union " + g2.label());

	for(var i=0; i<ng; i++)
	{
		g.addVertex();

		for(j=i+1; j<ng; j++)
		{
			if((n1 >= j && g1.hasEdge(i, j)) ||
			  (n2 >= j && g2.hasEdge(i, j)))
			{
				g.addEdge(i, j);
			}
		}
	}

	return g;
}
```

Simple edge-intersection

```
function graphIntersection(g1, g2)
{
	var n1 = g1.vertexCount();
	var n2 = g2.vertexCount();
	var ng = Math.min(g1, g2);

	var g = new Graph(g1.label() + " intersect " + g2.label());

	for(var i=0; i<ng; i++)
	{
		g.addVertex();

		for(j=i+1; j<ng; j++)
		{
			if(g1.hasEdge(i, j) && g2.hasEdge(i, j))
			{
				g.addEdge(i, j);
			}
		}
	}

	return g;
}
```

Randomize arc directions

```
function randomizeArcDirections(g)
{
	for(var a=0; a<g.arcCount(); a++)
	{
		if(Math.random() > 0.5)
		{
			g.flipArc(g.getArc(a));
		}
	}
}
```

BIBLIOGRAPHY

ECMA Script

Crockford, Douglas. *JavaScript: The Good Parts*. Yahoo Press, 2008.

ECMA International. *Standard ECMA-262: ECMAScript Language Specification (5th Edition)*. 2009. http://www.ecma-international.org/publications/standards/Ecma-262.htm.

Flanagan, David. *JavaScript: The Definitive Guide*. O'Reilly Media, 2006.

Mozilla Developer Network. "JavaScript Guide." MDN Doc Center. 2011. https://developer.mozilla.org/en/JavaScript/Guide.

Mozilla Developer Network. "JavaScript Reference." MDN Doc Center. 2011. https://developer.mozilla.org/en/JavaScript/Reference.

Graph Theory

Bondy, J.A., and U.S.R. Murty. *Graph Theory*. New York City: Springer, 2008.

Gross, Jonathan L., and Jay Yellen, eds. *Handbook of Graph Theory*. Boca Raton: CRC Press, 2003.

Graph Theory Algorithms

Habib, Michel, Ross McConnell, Christophe Paul, and Laurent Viennot. "Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition, and consecutive ones testing." *Theoretical Computer Science*. 234 (2000): 59–84.

Licenses

"Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0)." Creative Commons. http://creativecommons.org/licenses/by-sa/3.0/.

"GNU General Public License, version 2." Free Software Foundation. http://www.gnu.org/licenses/old-licenses/gpl-2.0.html.

"GNU Lesser General Public License, version 2.1." Free Software Foundation. http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html.

APPENDIX

Table 1. Existing graph theory software.

Title	URL	Descriptions taken from websites
Maxima	maxima.sourceforge.net/docs/manual/en/maxima_56.html	Free computer algebra system
Gato	gato.sourceforge.net/	a software which visualizes algorithms on graphs
js1k 618	js1k.com/demo/618	#js1k demo submission by Lars Rönnbäck
stixar-graphlib	code.google.com/p/stixar-graphlib/	A java library of efficient graph algorithms
AbuGraph	abugraph.sourceforge.net/	a Java application for graph visualization
OGDF	www.ogdf.net/doku.php	a self-contained C++ class library for the layout of diagrams
aiSee	www.aisee.com/	state-of-the-art graph layout software
annas	code.google.com/p/annas/	a Java framework designed for researchers in Graph Theory
BabelGraph	www.babelgraph.org/index.html	graph tool that provides an interactive and intuitive experience
CCVisu	www.sosy-lab.org/~dbeyer/CCVisu/	tool for Visual Clustering and Force-Directed Graph Layout
uDraw(Graph)	www.informatik.uni-bremen.de/uDrawGraph/	a professional graph drawing solution
Graph Magics	www.graph-magics.com/	an Ultimate tool for graph theory
Graphviz	www.graphviz.org/	open source graph visualization software
GRINEdit	www.nishiohirokazu.org/grinedit/	an open source visualization software for Graphs
gteditor	code.google.com/p/gteditor/	A graph editor for graph theoretical problems
GUESS	graphexploration.cond.org/index.html	data analysis and visualization tool for graphs and networks
HyperGraph	hypergraph.sourceforge.net/	java code to work with hyperbolic geometry
linloglayout	code.google.com/p/linloglayout/	Graph clustering and force-directed graph layout
Mascopt	www-sop.inria.fr/mascotte/mascopt/	a set of tools for network optimization problems
JGraph	www.jgraph.com/jgraph.html	the leading Java Swing Graph Visualization and Layout library
P.I.G.A.L.E.	pigale.sourceforge.net/	Public Implementation of a Graph Algorithm Library and Editor
Tom Sawyer	www.tomsawyer.com/home/index.php	Graph visualization
Tulip	tulip.labri.fr/TulipDrupal/	framework for analysis and visualization of relational data

Title	URL	Descriptions taken from websites
Walrus	www.caida.org/tools/visualization/walrus/	a tool for visualizing large graphs in three-dimensional space
Gephi	gephi.org/	an interactive visualization and exploration platform for graphs
yEd	www.yworks.com/en/products_yed_about.htm	diagram editor that can be used to quickly generate drawings
Sage	www.sagemath.org/doc/constructions/graph_theory.html	A free mathematics software system licensed under the GPL
UbiGraph	ubietylab.net/ubigraph/	a tool for visualizing dynamic graphs
very_nauty lib	keithbriggs.info/very_nauty.html	C library of graph algorithms
graphalg	sourceforge.net/projects/graphalg/	Software for Teaching Graph Theory Algorithms
GDToolkit	www.dia.uniroma3.it/~gdt/gdt4/index.php	An object-oriented C++ library for handling and drawing graphs
NetworkX	networkx.lanl.gov/	High productivity software for complex networks
yFiles	www.yworks.com/en/products_yfiles_about.htm	Java library for analysis, visualization, and layout of graphs
JGraphT	jgrapht.sourceforge.net/	mathematical graph-theory objects and algorithms
GOBLIN	goblin2.sourceforge.net/	Goblin is a full-featured tool chain for handling graphs
GraphThing	graph.seul.org/	a tool that allows you to create, manipulate and study graphs
Maple	www.maplesoft.com/support/help/Maple/view.aspx?path=GraphTheory	routines for creating, drawing, manipulating, and testing graphs
Mathematica	reference.wolfram.com/mathematica/tutorial/GraphDrawingIntroduction.html	functions for the aesthetic drawing of graphs
grTheory	www.mathworks.com/matlabcentral/fileexchange/4266	28 functions for different tasks of graph theory
nauty	cs.anu.edu.au/~bdm/nauty/	a program for computing automorphism groups of graphs
graph-utils	sourceforge.net/projects/graph-utils/	utilities realizing theoretical graph algorithms