Carleton University
School of Computer Science
Comp 5901 Directed Studies

JGraphEd – A Java Graph Editor and Graph Drawing Framework

Author: Jon Harris
Submitted: April 28, 2004
Submitted To: Professor Pat Morin

Table of Contents

Table of Figures

1 Introduction

This document describes the design and implementation of JGraphEd, a Java Graph Editing application and Graph Drawing framework. JGraphEd was designed to allow users to easily create graphs step by step by adding or removing, or modifying nodes or edges. JGraphEd is modular by design and a variety of standalone and interdependent algorithms have been provided for manipulating or visualizing graphs. JGraphEd was also designed with extensibility in mind, in order to allow developers to quickly and painlessly add their own algorithms to the included library.

Section 2 of this document describes the framework of JGraphEd. This includes classes which are responsible for representing graphs and the commands that can be used upon them. The framework documentation also describes the user interface of JGraphEd, and may be used as a user's guide.

Section 3 of this document gives a description of the various data structures that are incorporated into JGraphEd. Following the description of each data structure are references to where it was used in JGraphEd, either in the framework, or in operations which are packaged with it.

Section 4 provides a guide on how to incorporate new graph operations or algorithms into JGraphEd. The rest of this section gives a fairly extensive description of each of the operations or algorithms that are packaged with JGraphEd. The descriptions of these operations can be a valuable reference to developers intending to develop their own operations.

2 Framework

This section describes the core functionality of JGraphEd that makes up the framework for creating graphs and subsequently using these graphs as input to any number of graph algorithms.

2.1 Graph Structure

Fundamental to the functionality of JGraphEd is its representation of Graphs, and associated Nodes and Edges. The graphStructure package contains all of the classes used to represent these objects in JGraphEd. The following sections will describe and motivate the use of these classes.

2.1.1 Graph

The Graph class is responsible for storing a list of the nodes that it contains, providing an interface for editing parts of the graph, providing a centralized means for drawing all parts of the graph, tracking undo and redo actions and maintaining a log of algorithms applied to it.

2.1.1.1 Interface for editing

The Graph class uses the Façade pattern [GHJ+94] to restrict users from editing the parts of the graph directly. This allows the graph to update undo or redo information whenever a part of the graph is changed (if undos are being tracked). This also facilitates the determination of whether or not the graph needs to be redrawn as a result of changes to its parts.

The editing interface allows several editing operations to be performed. Below is an abbreviated list of these operations:

- Add / remove Nodes
- Add / remove Edges to/from a Node (with options for preventing duplicates)
- Flag Edges as 'Generated' edges created by an algorithm rather than by the user.
- Change the drawing colour of Nodes or Edges.
- Attach or edit text labels to be drawn next to Nodes.
- Curve, orthogonalize (bend) or straighten Edges.
- Select / unselect groups of or individual Nodes or Edges. (stored in a list of selected nodes or edges)
- Scale, rotate or translate all or parts of the Graph.
- Perform many of the above operations on groups of selected nodes or edges, or generated edges.
- Save / Load Graphs to / from files.

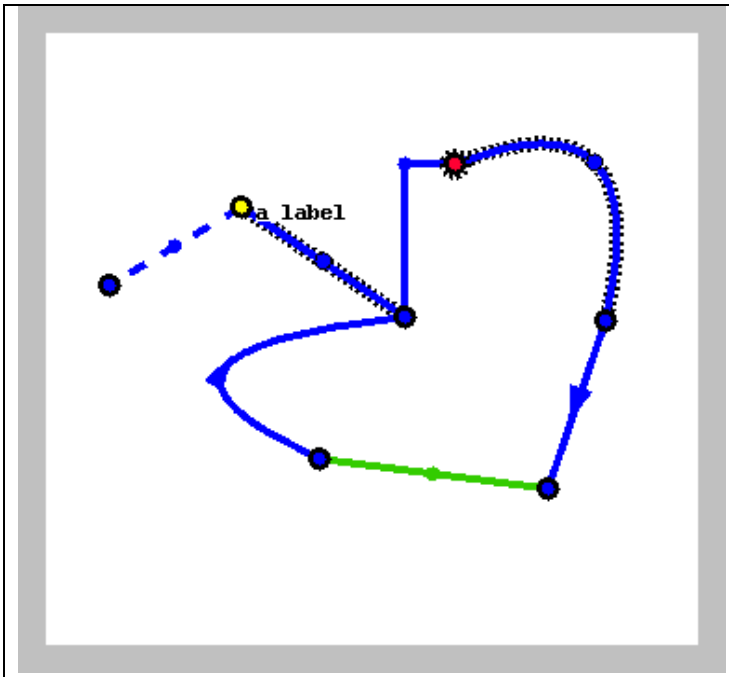

Figure 1 A sample graph being edited with JGraphEd

This figure shows off some of the features of JGraphEd.

- Generated edges are drawn dashed.
- Selected Nodes and Edges are surrounded by black-dashed outlines.
- The colour of nodes and edges can be set to any value.
- Edges can be curved, or made orthogonal (bent at right angles).
- Edges can be directed from one end node to another (denoted by an arrow)
- Labels can be attached to nodes.

2.1.1.2 Drawing

The Graph class provides a drawing method which draws all of the composite Nodes and Edges to the user interface. The draw method can make use of a set of optional parameters:

- X and Y offset to shift the location of drawing (allows borders to be displayed)
- Global colour to override the individual colours of composite Nodes and Edges.
- Number of rows and columns and corresponding width and height of a grid.
- Flag specifying whether or not Node labels, coordinates or neither should be drawn.

The drawing of the graph is saved to an image object. Calls to the draw method of the graph cause this image to drawn to the user interface. When the draw method is called, the graph verifies whether any changes have been made (through the editing interface) that require the image to be updated. If changes have been made since the last image was created, the graph re-draws itself to the image prior to drawing the image to the screen.

One benefit of this ‘buffering’ approach is that operations such as scaling, translating and rotating the graph can be displayed on the screen using corresponding

image transformations which are significantly faster than redrawing the entire graph from scratch.

2.1.1.3 Log Entries

The Graph class maintains a list of log entries pertaining to operations (algorithms) that have been executed upon it. A log entry includes the name of the operation, the number of nodes and edges that were added, the time taken, and a list of log entries pertaining to operations used as subroutines. Log entries may also store a text string providing additional information specific to the operation.

2.1.1.4 Undo / Redo Mementos

The Graph class maintains a doubly linked list of mementos objects [GHJ+94] which store information for undoing or redoing operations which have been performed upon it.

The package graphStructure.mementos contains a set of classes defining the mementos available in JGraphEd. All mementos must implement the MementoInterface, which provides the means for applying a memento, and determining whether or not it had any effect (allowing redundant undos/redos to be suppressed). Memento objects (implementing the Memento Interface) are responsible for defining the functionality for applying themselves to the graph. Memento objects can be applied in forward or reverse for redo or undo capabilities. The MementoGrouper class allows a sequence of mementos to be grouped together when several sub-operations are performed by a single operation.

When the user undoes or redoes an operation on the graph, the current position reference into the doubly linked list is decremented or incremented. Whenever an operation is performed on the graph, a memento is added at the subsequent position to the current position reference. This allows for an unlimited (subject to memory availability) number of undos and redos to be performed. It should be noted however that undoing an operation and then performing another operation prevents the undone operation from being redone.

2.1.1.5 Copies

An important feature of JGraphEd is the ability to make identical copies of graphs. This feature can be critical to graph algorithms that wish to make changes to a copy of the graph without affecting the original. When making a copy of a graph, care was taken so that nodes and edges occur in the same order in the copy as in the original.

The Graph class provides a variety of copy methods. Copies can be made of the entire graph, or copies can be made based on a subset of the nodes or edges in the

original. Optionally, the nodes and edges in the copy of the graph can each maintain references to the corresponding node or edge in the original graph. If a copy of a copy of a graph is made, each node or edge can traverse these references to determine the 'master original' node or edge.

2.1.2 Node

The Node class is responsible for storing its location and for maintaining a list of the edge objects that are incident to it. In order to allow constant time insertion or removal of edges at a node, the edges are stored as a doubly linked list. Each node also stores the number of edges incident to it, which allows the graph to count the number of edges it contains via a single traversal of the nodes.

The location of the node is stored in a location object, which stores real-valued (non-integer) x and y coordinates of the node. Using real-valued coordinates helps prevent rounding errors when the node is relocated by an operation such as rotation. The location object provides methods for retrieving either the real-valued coordinates, or rounded integer values of the real-valued coordinates.

The Node class provides methods for inserting edges at arbitrary or specific positions in the doubly linked list. Optionally, these methods can check for the existence of duplicate edges, preventing insertion of duplicates, and thus keeping the graph simple. The Node class also provides methods for removing specific edges, or returning an iterator object [GHJ+94] that allows ordered traversals of the edges incident to the node.

In addition to storing a location and list of edges, each node stores a label, colour and selection flag which can be used to decide how to draw the node to the user interface.

The Node class also provides helper methods used by its containing Graph's editing interface for rotating the node around a point, and scaling or translating it to a new location.

2.1.3 Edge

The Edge class is responsible for storing the two node objects that it is incident to, and information pertaining to whether or not it is curved, orthogonal or straight, and/or directed. The Edge class also provides methods for determining the length of an edge.

Rather than directly storing references to its two end nodes, each edge stores two references to instances of HalfEdge class, one for each of the end nodes. Each half edge stores a reference to its parent edge, an end node, its twin half edge (the other half edge belonging to its parent edge), the next half edge in order from its twin's end node, and the previous half edge in order from its end node.

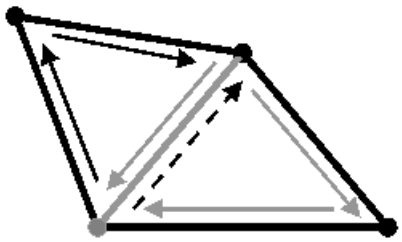

Figure 2 Half-Edge Structure

Illustration of the data stored (shown in grey) by a half edge (shown in dashed-black).

The use of this 'half edge' data structure aids in the maintenance of the doubly linked lists of edges at each of the nodes. Besides allowing in-order traversals of all the edges at a node, this structure also conveniently allows for an in-order traversal of all the edges around any face in the graph.

As mentioned previously, edges may be curved, orthogonal, and straight and/or directed. Curved or orthogonal edges are represented through the use of a center point location stored at each edge. For straight edges, this center point has coordinates corresponding to the average of the x and y coordinates of the two end nodes. For orthogonal edges, the center point has x and y coordinates corresponding to either the x coordinate of the first end node, and y coordinate of the second end node, or vice versa.

The case for curved edges is somewhat more complicated due to JGraphEd's use of Bézier curves [Bou96]. A Bézier curve allows for a segment of a quadratic curve to be specified using 3 control points P_0, P_1, and P_2. The Java Programming language provides built in support for drawing Bézier curves using the 3 control points, however it would be somewhat awkward for the user to have to drag the point P_1 around to change the shape of the curve. A more desirable approach would be to allow the user to drag around a point P_c that is actually ON the curve. For this reason, JGraphEd performs a mapping between the center point location P_c of the edge, and the second control point P_1 of the Bézier curve. Using the properties of Bézier curves, and some basic algebra, it is possible to determine the $P_c = ¼ P_0 + ½ P_1 + ¼ P_2$. The edge class has a method called getQuadCurve which performs this mapping prior to drawing the edge to the screen. It should be noted that when a Bézier curve is created, JGraphEd preserves the angles between P_c and P_0, and P_c and P_2, when the end-points are relocated in any way. This preserves the 'shape' of the curve under these operations. In order to compute the length of a curved edge, JGraphEd uses the properties of Bézier curves to approximate the length. A fixed number of points along the Bézier curve are sampled, and the sum of the Euclidean distance between these points is returned.

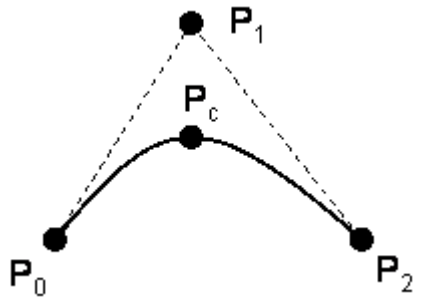

Figure 3 Bézier Curve and Control Points

Illustration of a Bézier curve (shown in solid black) and the 3 control points: P_0, P_1 and P_2.

The relationship between P_1 and P_c is also shown.

To allow it to be directed, each edge maintains a reference to the node which is the source for the direction. A directed edge is drawn to the user interface with a direction arrow which points away from the directed source node. The Edge class has a method called getDirectionArrow which use geometry to calculate the coordinates of the triangle which will be drawn to the screen.

2.1.4 Node and Edge Extenders

Many graph algorithms will require maintaining additional information about the nodes and edges in a graph in order to perform their function. Adding fields and methods to the node and edge classes for every such algorithm is cumbersome, and would quickly pollute these classes and break encapsulation.

To combat this problem, JGraphEd makes use of the decorator pattern [GHJ+94] to create NodeExtender and EdgeExtender classes. There is a cyclic relationship between nodes and node extenders and edges and edge extenders. For example, each node maintains a reference to its extender, and each node extender maintains a reference to its node. Nodes and node extenders implement the same interface, allowing them to be used interchangeably. This is also true for edges and edge extenders.

Using this scheme, in order to add fields or methods to nodes or edges, an algorithm need simply create its own sub-class of the NodeExtender and EdgeExtender classes. The graph class provides two key methods to support this scheme; The createNodeExtenders and createEdgeExtenders methods. These methods take a NodeExtender or EdgeExtender class as a parameter, and return a list of all the nodes or edges in the graph with instances of the provided class attached to each.

2.1.5 Graph Files

As mentioned previously, graph objects provide methods for loading and saving themselves to graph files. Graph files are a format specific to JGraphEd that preserve all of the information about the graph, including node/edge ordering, colours, node labels,

edge curves, edge orthogonality, edge direction etc… It is worth noting that JGraphEd also supports saving graphs to gif or jpeg image files.

Below is a brief description of the syntax of JGraphEd's .graph file format:

<- Graph Label ->
<- Number of Grid Rows ->
<- Width of each Grid Row ->
<- Number of Grid Columns ->
<- Width of each Grid Column ->
<- Number of Nodes ->
<- List of Nodes ->
    <- Node Index ->
    <- Node Real-Valued X Coordinate ->
    <- Node Real-Valued Y Coordinate ->
    <- Node Label ->
    <- Node Colour as RGB Integer ->
    <- List of comma delimited Incident Edges Indices ->
<- Number of Edges ->
<- List of Edges ->
    <- Edge Index ->
    <- First End Node Index ->
    <- Second End Node Index ->
    <- Directed Source Node Index, or -1 if none ->
    <- Center Location Real-Valued X Coordinate ->
    <- Center Location Real-Valued Y Coordinate ->
    <- Boolean for whether or not the Edge is Curved ->
    <- Boolean for whether or not the Edge is Orthogonal ->
    <- Boolean for whether or not the Edge is Generated ->
    <- Edge Colour as RGB Integer ->

2.2 User Interface

This section describes the user interface of JGraphEd, which allows the user to interact with the framework described in the previous section.

2.2.1 Menu and Toolbar

Perhaps the most predominant user interface component of JGraphEd is the menu and toolbar shown at the top of the screen. The menu and toolbar are linked using Java actions [Dav00] which allow menu items and toolbar buttons to be created simultaneously using the same data. In certain cases, toolbar buttons (and associated menu items) were grouped together to save space in the user interface. To provide this functionality, JGraphEd provides an action subclass called

CompoundMenuAndToolBarAction which groups together several regular actions. This new action subclass is used to instantiate a new UI widget created for JGraphEd called a ButtonChooser. The ButtonChooser creates a drop-down pane displaying several regular buttons that can be clicked. The corresponding menu items are displayed as radio menu items, which only allow for one menu item to be selected at a time.

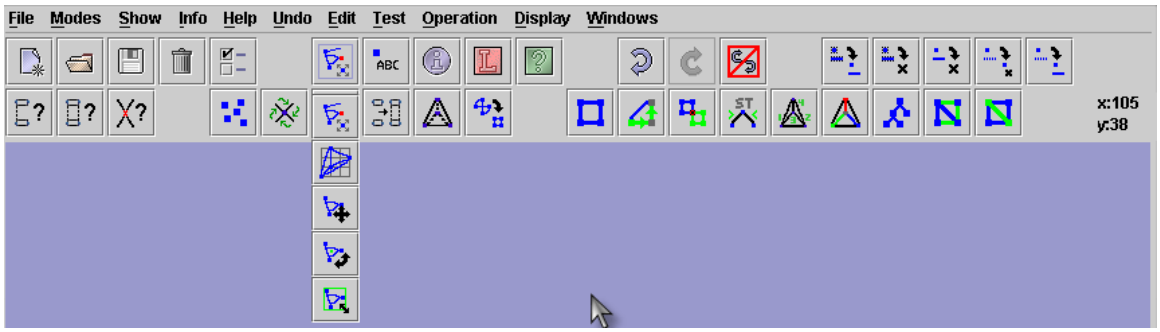

Figure 4 Snapshot of JGraphEd's menu and toolbar, with button chooser expanded/showing

JGraphEd stores all of the Actions, Buttons and Menu Items for the menu and toolbar in an instance of the MenuAndToolBar class. This class provides methods for populating the menu and toolbar, enabling or disabling subsets of controls as required, updating the currently displayed mouse cursor coordinates and updating the descriptions of the commands to be performed by the next undo or redo.

The loadControls method of the MenuAndToolBar class is responsible for determining what gets added to the menu and toolbar. This method creates and returns a hierarchical list of text strings describing all groups of commands, individual commands and sub-commands. The intent is to read these strings from a configuration file in a future release of JGraphEd.

All text strings for populating the menu and toolbar contain comma delimited values, the first value being a type identifier. The type identifier may have one of the following values:

M	Describes a menu header
S	Describes a single action for creating individual menu items and toolbar buttons
P	Describes a group of actions, for creating a group of radio button menu items and button chooser buttons.
separator	Denotes that a separator should be placed on the toolbar.
cursorLocationLabel	Used to determine the relative location of the cursor location label.

Text strings starting with the ‘M’ identifier provide values for the name of the menu, keyboard shortcut, and descriptive text.

Text strings starting with ‘S’ provide values such as the parent menu name, menu item text, toolbar button image, descriptive text, keyboard shortcut, initial enablement value, button width, and method (of the GraphController class described below) to execute when the button or menu item is clicked.

Text strings starting with ‘P’ are slightly different than those starting with ‘S’ in that the method to execute is replaced by the number of children. Text strings starting with ‘P’ are also longer than other descriptors. They are appended with text strings starting with the ‘C’ type identifier, specifying the actions which are children of the group. Text Strings starting with the ‘C’ type identifier follow the same syntax as those starting with the ‘S’ type identifier.

Separator type text strings must specify what type of separator to use; the options are ‘space’ and ‘newline’. The cursorLocationLabel type text string does not allow any options.

The cursor location label is used to display the current coordinates of the mouse cursor relative to the coordinate space of the graph currently being edited. The cursor location is updated continuously through the use of a separate thread.

2.2.2 Graph Editor

In order to allow the user to create and modify graph objects (and all of their composite parts), JGraphEd uses an instance of the GraphEditor class. The graph editor is responsible for displaying current graph editing actions to the user. For example, when the user is creating an edge, the graph editor will draw a line from the starting node to wherever the user drags the mouse.

Other responsibilities of the graph editor include:

- Drawing other shapes such as the center point when the graph is being rotated, or the bounding box when it is being scaled.
- Changing the mouse cursor when the user is performing an action such as rotating the graph.
- Storing an image that is created when the underlying graph is drawn.
- Determining when the graph should be redrawn from scratch, or when image transformations can be applied to the image to speed up the refresh rate (such as when the entire graph is rotated, scaled, or moved).
- Storing special node selections for use as input to operations (can be an arbitrary number of nodes, or 3 nodes bounding a face etc.)

JGraphEd allows multiple graph editors to be opened simultaneously, to allow for concurrent editing of several different graphs. This was accomplished using Java's InternalFrame and JDesktopPane classes [Unk04].

2.2.2.1 Graph Editor Modes

The handling of user events that are performed on the canvas of the graph editor, such as clicks, drags, and key presses is delegated to an instance of the GraphEditorListener class (or subclass) which is attached to the graph editor. A graph editor can operate in one of 5 different modes: edit mode, grid mode, move mode, resize mode and rotate mode. A sub-class of the GraphEditorListener class exists for each of these modes in the userInterface.modes package. These classes use Java's built-in event listeners to respond to mouse or keyboard actions performed by the user inside the graph editor's drawing canvas. Subclasses of the GraphEditorListener are the controller of the Model-View-Controller pattern [BMR+00], where the Graph class is the model, and the GraphEditor class is the view.

The GraphEditor class provides methods for switching its editing functionality between these 5 different modes. The following sections describe the features of the 5 different modes.

2.2.2.1.1 Edit Mode

The edit mode is designed to provide the user with a wide array of options for creating and editing graphs. The edit mode makes use of a so-called 'node split tree' which is a rudimentary implementation of a KD-Tree (see section 3.4) to speed up the finding of nodes at a specific editing location. Without this data structure, the edit mode would have to search through all the nodes whenever the mouse was clicked (or dragged when creating an edge) to determine whether a node exists at that location. Below is a table of all the available commands for edit mode and their triggering and resulting actions.

Command	Trigger	Resulting Action
Node creation	Single / Double click in empty space	A new node is created at the current mouse location
Node Selection	Single Click on an existing node, or drag the mouse from an empty location to create a rectangle around the desired nodes.	The selection of the node is turned on (black dashed circle around) or off.
Node Movement	Press and drag the mouse over a selected node. (Holding down control moves all selected nodes)	The selected node is dragged to a new location.
Node Deletion	Press the delete key or use the toolbar or menu buttons.	All selected nodes (and edges) are deleted

Node Labelling	Right-Click a node a select the 'label' button from the popup menu (Holding down control before right-clicking re-labels all selected nodes)	The node label is changed to the text entered. (note: Select 'show labels' from the menu or toolbar if labels are not visible)
Node Colouring	Right-Click a node and select the 'colour' button from the popup menu (Holding down control before right-clicking re-colours all selected nodes)	The node colour is changed to the value that is selected.
Edge Creation	Press and drag the mouse over a non-selected node, and drag to the desired end location, and release.	An edge will be created to the end location. (A preference can be set to determine if a new node should be created at the release location)
Edge Selection	Single-click the 'bump' located at the center of the desired edge, or drag the mouse from an empty location to create a rectangle around the desired edges.	The selection of the edge is turned on (black dashed line enveloping) or off.
Edge Deletion	Press the delete key or use the toolbar or menu buttons.	All selected edges (and node) are deleted.
Edge Order Display	Press and Hold the control key, and then click on a node, or center 'bump' of an edge. (Holding down control-shift shown the order in reverse for nodes)	The edges around the chosen node are flashed in order, or the edges bordering the same faces as the chosen edge are flashed in order.
Edge Curving	Select the edge to curve, and then click and drag its center point (bump).	The edge will be curved using the new center location as a control point for a Bézier curve.
Edge Straightening	Right-click on an edge (center-point or bump) and click the 'make straight' button (Holding down control straightens all selected edges)	The edge(s) will be straightened, so that its center point is aligned with its end nodes.
Edge Directing	Press and drag the mouse from the source end node to the destination end node.	The edge is directed (arrow pointing in direction) towards the destination node.
Edge Un-Directing	Press and drag the mouse from the destination node to the source node, or right click and click 'remove direction' from the popup menu. (Holding down control before right-clicking will un-direct all selected edges)	The direction of the edge is removed

Edge Colouring	Right click an edge (center-point) and click the 'colour' button from the popup menu (Holding down control will change the colour of all selected edges)	The colour of the edge will be changed to the selected value.

2.2.2.1.2 Grid Mode

The grid mode is similar to the edit mode, except nodes are restricted to being created at or dragged to intersections of grid lines drawn to the screen. When the user switches to grid mode, they are prompted for the number of rows and columns in the grid, and the width and height of the grid cells. The following table follows the format of the table for edit mode, except that only commands which are different from those of the edit mode are shown.

Command	Trigger	Resulting Action
Node creation	Single / Double click in empty space	A node is created at the closest grid intersection to the mouse location.
Node Selection	Single Click on an existing node, or in the grid cell near to it, or drag the mouse from an empty location to create a rectangle around the desired nodes.	The selection of the node (or node at closest grid intersection) is turned on (black dashed circle around) or off.
Node Movement	Press and drag the mouse over a selected node, or right click a node NOT at a grid intersection point, and choose 'snap to grid' from the popup menu. (Holding down control DOES NOT moves all selected nodes)	The selected node is dragged to a new location at the closest grid intersection point to the mouse cursor.
Edge Orthogonalization	Select the edge to curve, and then click and drag its center point (bump) to either side of the edge, or right-click a curved edge, and choose 'make orthogonal' from the popup menu (This replaces Edge Curving from Edit Mode)	The edge is bent to form a 90 degree angle on the side of the edge that the mouse is dragged to.

2.2.2.1.3 Move, Resize and Rotate Modes

The move, resize and rotate modes provide functionality for translating, scaling or rotating the graph respectively. These are the only functions that these modes allow the user to perform.

Move mode changes the mouse cursor to a hand icon, and translates the entire graph whenever the user presses and drags the mouse. Resize mode changes the mouse cursor to resize icons whenever the user moves the mouse over the bottom and/or right edges of the displayed bounding box, and scales the entire graph whenever the user presses and drags the mouse. Rotate mode changes the mouse cursor to a turning icon, displays a pivot point at the graph's center of gravity, and rotates the entire graph whenever the user presses and drags the mouse.

It should be noted that resizing the graph to a very small size, and then resizing it back to the original size may result in some nodes becoming bunched together near the top and/or left boundary of the graph due to round-off errors.

2.2.2.2 Graph Editor Dialogs

A graph editor may require the use of dialog boxes to acquire specialized or operation-specific information from the user. To meet this requirement, each graph editor maintains a reference to a single instance of the GraphEditorDialog class. Creating a subclass of this class allows for dialogs to be displayed that are tailored to the needs of a specific operation. JGraphEd was designed so that dialogs are modal with respect to their associated graph editor. For example, if the user clicks a toolbar button which opens a dialog, and then switches to another graph editor, they will be forced to deal with that dialog whenever they switch back to the original graph editor. Closing a graph editor will also close any of its associated dialogs.

2.2.2.3 Graph Editor Info Window

Each graph editor maintains a reference to an instance of the GraphEditorInfoWindow class. This class is responsible for showing helpful information about the graph being edited, such as the number of nodes and edges. The information displayed in the info window is updated every time a change occurs to the graph being edited by its associated graph editor. For performance reasons, it may be unwise to leave a graph editor's info window open if the edited graph has many nodes and/or edges. Closing a graph editor will close its associated info window, if it is visible.

2.2.2.4 Graph Editor Log Window

Each graph editor maintains a reference to an instance of the GraphEditorLogWindow class. This class is responsible for providing a hierarchical (tree) view of all of the log entries stored by the graph being edited by its associated graph editor. Whenever a new logged operation is performed on the graph, the log

window is update with a new entry. New entries may contain sub-entries for logged operations that were used as sub-routines for the main operation. These sub-entries may be examined by expanding the hierarchical view of the main log entry using the appropriate icons. Closing a graph editor will close its associated log window, if it is visible.

2.2.3 Graph Controller

The GraphController is the mediator [GHJ+94] which serves as an intermediary between operations and individual graph editors on which they are to be performed. The GraphController maintains a reference to the currently active graph editor. Whenever a new graph editor becomes active, the GraphController is responsible for updating the menu and toolbar to reflect the new graph editor's state. The menu and toolbar is responsible for providing menu items or toolbar buttons which may be used to invoke operations. This is accomplished by defining methods in the GraphController class, which are called by the MenuAndToolbar class. These methods in turn invoke the appropriate method of the appropriate operation class.

2.2.3.1 Graph Editor Preferences Window

The GraphController class maintains a reference to an instance of the GraphEditorPreferencesWindow class. This class is responsible for providing a centralized location for the user to set a variety of preferences that affect how JGraphEd functions. Below is a table of the preferences that can be modified by the preferences window:

Preference	Description
Single Click to Add Nodes	When enabled, nodes can be created with a single or double click. When disabled, nodes must be created with a double click.
Create a New End Node if a new Edge has only one Node	When enabled, while the user is creating an edge, they may release the mouse over empty space, and a new end node will be created at that location, connected by the new edge. When disabled, users can only create edges between existing nodes (unless the control-key is held down)
Default for Draw Canonical Ordering and Normal Labeling on Embedding	When enabled, the default value for displaying the canonical ordering or normal labeling of a graph will be to display it after it has been straight-line grid embedded. When disabled, the default will be not to straight-line grid embed first.
Clear Generated Edges	When enabled, all generated edges (by triangulation etc…)

after Straight Line Embedding	will be removed after the graph is straight-line grid embedded. When disabled, these edge will not be deleted.
Draw Text Background on Top of Edges	When enabled, the background of the bounding box around the label text for each node will be drawn opaquely after all edges have been drawn. When disabled, the label text for each node does not obscure the edges, but is much harder to read.

2.2.3.2 Graph Editor Help Window

The GraphController class maintains a reference to an instance of the GraphEditorHelpWindow class. The graph editor help window provides help and usage information to users of JGraphEd, based on a set of HTML files. The graph editor help window provides a primitive HTML browser for browsing these help files, as well as a content tree and buttons for navigating through them.

3 Data Structures

JGraphEd provides a variety of data structures for use either in its framework, or in operations that extend it. The corresponding classes are located in the dataStructure package.

3.1 Doubly Linked List

The DoublyLinkedList class provides an implementation of the standard doubly linked list [CLR90]. This class supports the constant time insertion or deletion of elements, and allows for forward or reverse traversals. Elements can also be inserted or removed in constant time based on the current position of a traversal.

At present, doubly-linked lists are used for storing and traversing mementos for undo and redo (Section 2.1.1.4), and for storing and traversing the history of help pages viewed by the user in the help window (Section 2.2.3.2).

3.2 Queue

The Queue class provides an implementation of the standard FIFO (First In, First Out) queue [CLR90]. This class supports the constant time insertion or deletion of elements. Inserted items are always placed at the end of the queue, and deleted items are always removed from the front of the queue.

At present, queues are used for storing PQ-Nodes during the preliminary phase of the reduction operation of PQ-Trees (Section 3.5).

3.3 Binary Heap

The BinaryHeap class provides a pointer-based implementation of the standard Binary Heap [CLR90]. The heap is a min-heap, meaning that operations are tailored towards extracting and returning the element with the smallest key. This class supports O(logn) time insertion, decreaseKey and extractMin operations when it contains n elements.

At present, binary heaps are used by the Dijkstra Shortest Path (Section 4.2.14), and Minimum Spanning Tree (Section 4.2.15) operations, allowing them to run in O(nlogn) time when performed on n nodes.

3.4 Node Split Tree (KD-Tree)

The NodeSplitTree class provides a rudimentary implementation of a KD-Tree [BSK+00]. A node split tree is initialized with the set of nodes from the graph, a maximum depth of the tree, and the radius of the drawing area of the nodes. During initialization, the node split tree repeatedly partitions the set of nodes recursively into 3 groups, keeping track of what level the partition occurred at. If the partition occurs at an even level, the nodes are partitioned around the node with the median X coordinate. If the partition occurs at an odd level, the nodes are partitioned around the node with the median Y coordinate. At a given level, the node split tree stores the median X or Y node, and passes the nodes with smaller or larger X or Y coordinates to initialize the next levels at its left or right children respectively.

The node split tree allows for queries to be performed for the existence of a node at a given location. When a query for a given location is performed, the node split tree tests whether a node stored at each level is within the drawing radius of that location. If the node is within the radius of the point, that point is returned, otherwise the query continues at either the left or right child of the node. The decision of which child (left or right) to forward the query to depends on whether the level is odd or even, and what the value of the location is. For even levels, if the location has an X coordinate smaller than that of the node, the query is forwarded to the left child, and if the location has an X coordinate larger than that of the node, the query is forwarded to the right child. For odd levels, the same process is used, except that Y coordinates are used instead of X coordinates.

It should be noted that if the maximum depth of the tree is less than log base 2 of the number of nodes in the tree, there will be lists of nodes stored at the leaves of the tree. In this case, queries which reach a leaf must scan all nodes in the list of that leaf. Therefore, the query time to find a node based on a location with a node split tree

containing n nodes with depth d is O(d + n /2 *d). For a depth of log (n), this results in a query time of O(logn) which is a significant speedup compared to O(n) time to search every node. There is however a price to pay for this node finding speedup; Building the node split tree requires O(n * d) time.

At present, a node split tree is used to speedup the finding of nodes at the current mouse location in edit mode (Section 2.2.2.1.1). Since the location of the nodes can be changed at any time in JGraphEd (requiring a rebuild of the node split tree), the depth of the tree is currently set to a default of 4 levels, allowing linear construction time.

3.5 PQ / PQD Tree

The PQTree class provides a full implementation of the PQ-Tree data structure developed by Booth and Lueker [BL76]. PQ-Trees are a data structure for representing the permutations of a set of elements, in which various subsets of the elements are constrained to occur consecutively. The fundamental elements of PQ-Trees are P-Nodes and Q-Nodes. P-Nodes allow their children to be permuted in any order, while Q-Nodes allow only a reversal of the fixed ordering of their children. Both P and Q nodes are represented by the PQNode class used by the PQTree class.

The only operation that can be performed on a PQ-Tree is the reduction operation. The reduction operation takes as input a subset of the elements of the PQ-Tree which are to be constrained to appear together. The reduction operation uses a series of template matchings and replacements, performed in a bottom up fashion until the lowest common ancestor of all the reduction elements is encountered. After a reduction has been performed, the number of possible permutations of the elements of the PQ-Tree is reduced, and the resulting PQ-Tree represents a class of permissible permutations of the elements. If the given reduction was not possible, an empty PQ-Tree is returned. The most desirable feature of PQ-Trees, is that with careful management, the reduction operation can be performed in time linear in the number of elements passed to it.

The PQ-Tree implementation provided by JGraphEd is augmented so that it is essentially a PQD-Tree [CAN+85]. A PQD-Tree is a PQ-Tree which allows an additional type of node, called a D-Node, which allows for the direction of the order of children of a Q-Node to be specified. A PQD-Tree operates in the same way as a PQ-Tree, in that D Nodes are essentially ignored or ‘passed over’ when performing a reduction operation. D Nodes are also represented by the PQNode class used by the PQTree class.

At present, the PQ-Tree data structure is used to implement a linear time (in the number of nodes) Planarity Testing operation (Section 4.2.7) for a graph. The PQD-Tree data structure is used to implement a linear time Embedding operation (Section 4.2.8) for a graph.

4 Operations (Algorithms)

JGraphEd provides a variety of operations or algorithms which can be applied to any graph opened by it. The operations package contains the classes responsible for performing these operations. The operations.extenders package contains the classes that sub-class the NodeExtender and EdgeExtender classes to provide the additional node and edge fields and methods required by these operations.

4.1 Adding a New Operation to JGraphEd

JGraphEd was designed with extensibility in mind in order to encourage developers to develop new operations to be performed on graphs. The goal of this section is to provide instructions to aid developers in quickly developing and deploying new operations for JGraphEd.

The process of extending JGraphEd can be divided into 3 principle stages; Developing and attaching extenders to the graph, developing the operation class, and integrating the operation into the user interface.

Developing and attaching extenders to the graph first requires creating sub-classes of the NodeExtender and EdgeExtender class which contain fields and methods specific to and required by the new operation. Once these subclasses have been implemented, they can be attached to the graph using the methods described in section 2.1.4. The operations.extenders package contains many examples that can be used as a guide to creating these subclasses.

Developing the operation class is typically accomplished by implementing a new class which provides static methods that take a graph and any other required data as parameters. These static methods should attach any required extenders to the graph prior to performing their function. The static methods should also ensure that the input graph conforms to the requirements of the operation. Instances of the GraphException class contained in the graphException package can be thrown by these methods to notify JGraphEd of invalid input, causing an error dialog to be shown to the user. Operation classes are free to invoke any of the graph editing operations provided by the graph object's editing interface (see section 2.1.1.1). The operations package contains many examples that can be used as a guide to creating new operation classes.

Integrating the new operation into JGraphEd's user interface is a 2 step process. First, a method should be added to the GraphController class that invokes the appropriate method of the new operation class. This method of the graph controller should provide any input required by the operation, and catch any instances of GraphException that are thrown, and display appropriate error dialogs to the user. Secondly, a new text string describing menu items and toolbar buttons for running the operation should be added to the constructor of the MenuAndToolBar class (see section 2.2.1). This text string should reference the method added to the GraphController class.

4.2 Existing Operations in JGraphEd

This section provides background information and descriptions pertaining to the operations that are currently provided with JGraphEd.

4.2.1 Create Random Graph

The create random graph operation is defined in the CreateRandomGraphOperation class. It is a primitive operation that distributes a number of nodes at random locations in the graph. The user is prompted for the number of nodes to create, and these nodes are subsequently distributed at uniform x coordinate intervals, and random y coordinates across the current graph editor canvas.

The intent is that the user may then create edges between these nodes using other operations such as ‘make connected’, ‘make biconnected’ or ‘make maximal’. These operations are described in subsequent sections of this document.

4.2.2 Depth First Search

The depth first search operation is defined in the DepthFirstSearchOperation class. This class implements the standard depth first search operation as described in [Eve79]. This operation builds a tree of the nodes, based on the order that they are visited by the search. The depth first search operation requires the following additional fields for the nodes and edges:

- Node – Integer, Depth First Search Number
- Node – Integer, Depth First Search Low Number
- Node – Node, Depth First Search Parent
- Edge – Boolean, Is Back Edge
- Edge – Boolean, Is Used (has been traversed)

These fields are defined in the DFSNodeEx and DFSEdgeEx extender classes. The depth first search number of a node is set incrementally as it is visited by the search. The low number of a node is the smallest depth first search number that can be reached by the node using non-back edges followed by at most one back edge. The depth first search parent of a node is the node that preceded (parented) that node in the search tree. The ‘is back edge’ flag is used to mark edges that are traversed during the search and result in the visitation of a node that was already searched. The ‘is used’ flag is used to keep track of the edges that have already been traversed during the search.

4.2.3 Connectivity

The ConnectivityOperation class provides a variety of methods for performing connectivity operations related to graphs, such as returning all of the connected sub-graphs of a graph, returning all of the nodes connected to a node, testing a graph for connectivity, and making a graph connected. A graph is connected if all of its nodes can be reached from every other node using a sequence of the graph's edges.

To get all of the connected sub-graphs of a graph, the Connectivity class makes repeated use of the depth first search node operation. The graph's copy method is invoked on all of the nodes visited by a depth first search from an arbitrary node. This process is then repeated using a depth first search from any node that was not visited by the last depth first search.

Returning all nodes connected to a node is a trivial operation that simply returns all of the nodes visited by a depth first search starting at that node.

Testing a graph for connectivity is also a trivial operation, it simply determines whether or not there is exactly one connected sub-graph of the input graph.

Making a graph connected involves retrieving all of the connected sub-graphs of the input graph. Each of the connected sub-graphs is linked with a new edge to the next connected sub-graph until all of the sub-graphs are linked.

4.2.4 Biconnectivity

The BiconnectivityOperation class provides a variety of method for performing biconnectivity operations related to graph, such as returning all of the biconnected sub-graphs of a graph, finding all separator nodes, testing a graph for biconnectivity and making a graph biconnected. A biconnected graph is a graph which does not contain any separator nodes. A separator node is a node whose removal from the graph results in the graph being split into two non-connected graphs.

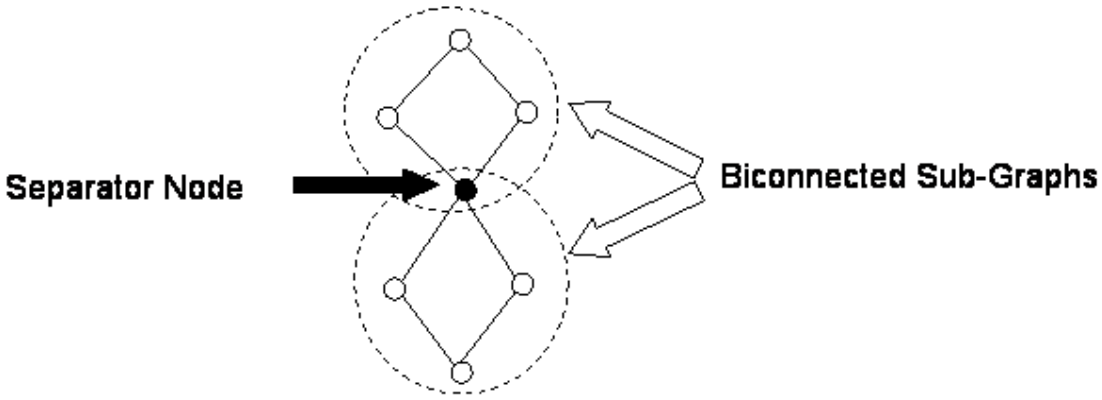

Figure 5 Illustration of Biconnected Sub-Graphs and Separator Nodes

The biconnectivity operation requires the following additional fields for the nodes and edges:

- Node – Integer, Depth First Search Number
- Node – Integer, Depth First Search Low Number
- Node – Node, Depth First Search Parent
- Node – Integer, Sub Graph Number
- Edge – Boolean, Is Back Edge
- Edge – Boolean, Is Used (has been traversed)
- Edge – Integer, Sub Graph Number

These fields are defined in the BiCompNodeEx and BiCompEdgeEx classes. The sub graph number fields are used to determine which of the biconnected sub-graphs each node or edge belongs to. It should be noted that the node and edge extender classes created for the biconnectivity operation are sub-classes of those created for the depth first search operation. The depth first search operation provides an option for reusing existing extenders rather than creating new ones. This allows the biconnectivity operation to share data with the depth first search operation.

To find all of the separator nodes, and biconnected sub-graphs of a graph, JGraphEd implements the linear time algorithm described in [Eve79]. This algorithm uses properties of the depth first search algorithm to find the separator nodes. The key property used is, that if and edge between nodes u and v is not a 'back edge', and u's depth first search number is greater than 1, and v's low number is greater than or equal to u's depth first search number, then u is a separator node. Also, if a node has a depth first search number of 1, and has at least two non-back edges, than that node is a separator node.

Testing a graph for biconnectivity is a trivial operation, it simply determines whether or not there is exactly one biconnected sub-graph of the input graph.

To make a non-biconnected graph into a biconnected graph, JGraphEd uses an algorithm developed by the author. This algorithm requires that the graph be connected (see section 4.2.3), and that the graph be embedded (see section 4.2.8), so that the order of the edges around each node is known. This algorithm runs in time linear in the number of nodes when the input graph is planar (the algorithm is only designed to be run on planar graphs).

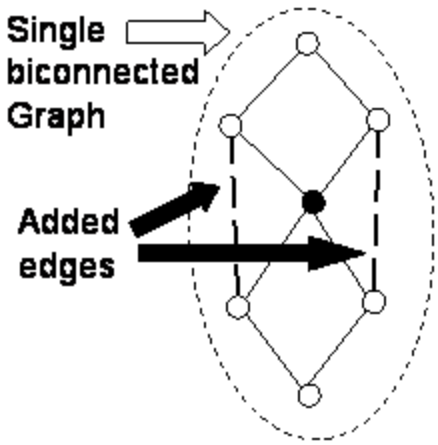

Figure 6 Illustration of Make Biconnected Operation

The algorithm for making a graph biconnected proceeds as follows:

- If the graph is not connected, connect it.
- Embed the graph.
- For each separator node of the graph:
    - For each pair of adjacent edges around the separator node:
        - If the two other end nodes of the adjacent edges are not connected by an edge and the two adjacent edges belong to different biconnected sub-graphs.
            - Add an edge between the two other end nodes of the adjacent edges.
            - If the separator node only has two incident edges
                - Break out of the loop after adding the first edge.

4.2.5 Make Maximal Planar

The MakeMaximalPlanar class provides methods for making a planar graph (see section 4.2.7) maximal planar. A maximal planar graph, sometimes referred to a triangulated planar graph, is a planar graph where the degree of every face is 3. In Other words, every face in the graph (including the outer face) is a triangle.

JGraphEd implements the algorithm described in [FPP90] for making a planar graph maximal. Although it is not stated in the paper describing it, the algorithm requires that the input planar graph be biconnected (see section 4.2.4).

The make maximal algorithm proceeds as follows:

- Make the graph biconnected (and embedded).
- For each node v:
    - For each pair of adjacent edges around v with end nodes u, w:

        - If the edge {w,u} does not immediately follow the edge {w,v} at w, then add the edge {u,w} between u and w.
- After processing all nodes, sort the edges lexicographically by end-node index to find duplicate edges. This can be done in linear time (in the number of nodes) by using radix sort combined with counting sort [CLR90].
- For each duplicate edge {x,y}:
    - Replace {x,y} by the edge {x',y'} which is the other diagonal of the (unique) quadrilateral {x,x',y,y'} which is the union of the two triangles adjacent to {x,y}.

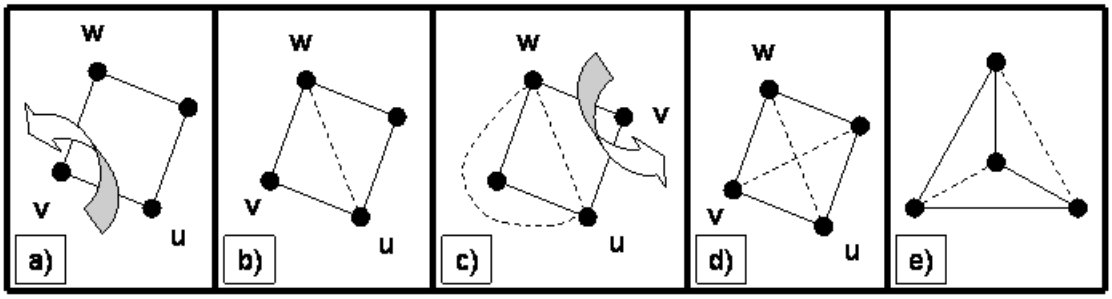

Figure 7 Illustration of Make Maximal Planar Operation

Figure 7 illustrates:

a) The input planar graph.
b) Edges added after processing the left-most node.
c) Edges added after processing the right-most node.
d) The corrected triangulation after flipping one of the duplicates.
e) The resulting triangulation.

4.2.6 ST Numbering

The STNumberOperation class provides methods for computing ST-numberings of the biconnected sub-graphs (see section 4.2.4) of the input graph. An ST-Numbering of a biconnected graph (which has no separator nodes) is a numbering of the N nodes of the graph that has the following properties:

- An arbitrary node has ST-Number 1.
- Some node adjacent to the node with ST-Number 1 has ST-Number N.
- Every node other than those with ST-Number 1 and ST-Number N has both adjacent nodes with a larger ST-Number and adjacent nodes with a smaller ST-Number.

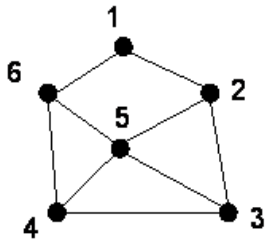

Figure 8 Illustration of an ST-Numbered Graph

JGraphEd implements the algorithm described in [Eve79] for computing an ST-Numbering of a biconnected graph in linear time (in the number of edges).

The ST-numbering algorithm first performs a depth first search starting at the node that is to have an ST-number of n. The first edge to be traversed will be the edge to the node having an ST-number of 1.

Secondly, the ST-Numbering operation is applied, using the depth first search number, low number, and tree parent computed by the depth first search. Prior to running this operation, all nodes and edges are marked as 'new'. The nodes first and last nodes (with ST-Number 1 and n), and the edge connecting them are marked 'old'.

The ST-numbering algorithm now proceeds as follows:

1. Place the nodes to have ST-Number 1 and n in a stack (1 on top of n).
2. Set i = 1
3. Remove a node v from the stack
4. If v is the node to have ST-Number n, then set its ST-Number to n and halt.
5. Apply the path-finding procedure to v.
6. If the path returned by the path-finding procedure is empty, then set the ST-Number of v to i, and increment i.
7. If the path returned is not empty, then insert all but the last node from the path on the stack in the reverse order of their occurrence on the path and go to step 3.

The path-finding procedure is defined as follows for a node v:

- If there is a 'new' back-edge e incident to v connecting to node w
    - Mark e as 'old' and return the path v->w
- If there is a 'new' tree edge e incident to v connecting to node w
    - Create a path p which starts with v, and ends with the node whose DFS number is equal to the low number of w.
    - Mark all edges along p as 'old' and return p.
- If there is a 'new' back edge e incident to v and connecting FROM node w
    - Create a path p which starts at v, and ends at the first 'old' node encountered.
    - Mark all edges along p as 'old' and return p.
- If there are no 'new' edges incident to v
    - Return an empty path.

The biconnectivity operation requires the following additional fields for the nodes and edges:

- Node – Integer, Depth First Search Number
- Node – Integer, Depth First Search Low Number
- Node – Node, Depth First Search Parent
- Node – Integer, ST Number
- Node – Boolean, Is Old
- Edge – Boolean, Is Back Edge
- Edge – Boolean, Is Used (has been traversed)
- Edge – Boolean, Is Old

These fields are defined in the STNodeEx and STEdgeEx classes. The purpose of these fields should be clear from the description of the algorithm. It should be noted that the node and edge extender classes created for the ST number operation are sub-classes of those created for the depth first search operation. The depth first search operation provides an option for reusing existing extenders rather than creating new ones. This allows the ST number operation to share data with the depth first search operation.

4.2.7 Planarity Testing

The PlanarityOperation class provides methods for determining whether the input graph is planar. A graph is defined to be planar if it can be drawn in such a way that none of its edges are crossing.

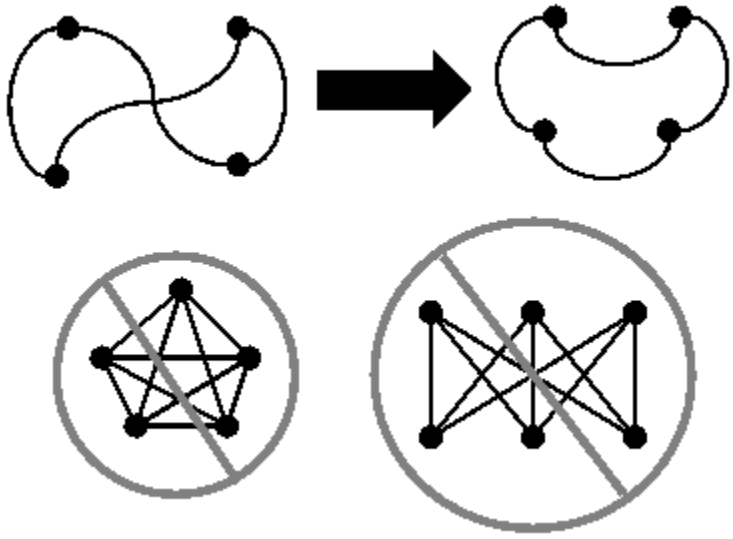

Figure 9 Illustration of Planar, and Non-Planar Graphs

JGrpahEd implements the algorithm for planarity testing described in [BL76]. This algorithm uses the PQ-Tree data structure (see section 3.5) to perform the planarity test in time linear in the number of nodes of the input graph.

This algorithm operates in three stages:

- Find the biconnected sub-graphs of the input graph.
- Computing an ST-Numbering of each of these biconnected sub-graphs.
- Perform reduce-and-replace cycles on each ST-numbered sub-graph using a PQ-Tree.

The planarity test works as follows:

- Make a PQ-Tree whose leaves are the set of all edges whose lower ST-Numbered node is 1.
- From j = 2 to N-1
    - Reduce the PQ-Tree, such that all leaves whose **higher**-numbered node is j appear consecutively.
    - If the reduction is impossible
        - The Graph is not planar
    - Otherwise
        - Replace the sub-tree with leaves whose **higher**-numbered node is j with a new sub-tree with leaves whose **lower**-numbered node is j.
- The graph is planar

The planarity testing operation requires the following additional fields for the nodes and edges:

- Node – Integer, ST Number
- Edge – PQNode, P or Q Node storing the edge in the PQ-Tree.

These fields are defined in the PQNodeEx and PQEdgeEx classes. The purpose of these fields should be clear from the description of the algorithm. This operation provides a good example of how values can be copied from an old set of extenders to a new set of extenders (this is done for each node extender's ST Number field).

4.2.8 Embedding

The EmbedOperation class provides methods for performing an embedding of a planar input graph. An embedding of a planar graph is a specification of an ordering of the edges around each node. More specifically it provides the order of edges around each node such that drawing the edges in this order for all nodes will result in a crossing-free drawing.

JGraphEd implements the linear time algorithm for computing an embedding of a planar graph as described in [CNA+85]. This algorithm uses PQD-Trees. It is similar to the planarity test, except that at the step in the planarity test where leaves are replaced, they are now stored as part of an *upward embedding* of the graph.

The purpose of the D-Nodes in PQD-Trees is to track the order in which the replaced edges are read from the PQD-Tree during the embedding process. Whenever the replaced edges are descendants of a Q-Node, a D-Node is added as its child pointing in the direction along which the children were removed. This is necessary, because it is not always possible to tell whether the descendants of a Q-Node are in left to right or right to left order.

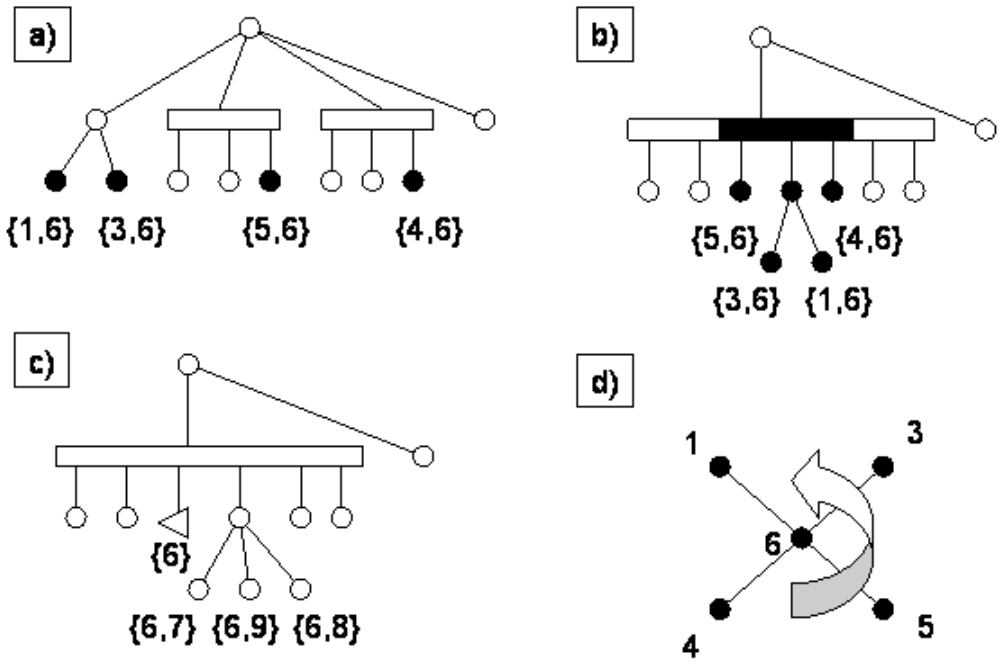

Figure 10 Illustration of the Embedding Operation

Figure 10 illustrates:

a) PQ-Tree before reduction.
b) PQ-Tree after reduction.
c) PQ-Tree after replacement and corresponding D-Node addition.
d) The upward embedding for the node at this step.

Removing D-Nodes from the PQD-Tree at subsequent iterations allows for corrections of the left-to-right order of the edges around the previous nodes. In other words, the D-Nodes ensure that for every node, the direction of the order of its edges is the same as that of every other node (all clockwise or all counter-clockwise).

A final phase of this algorithm extends the upper embedding of the graph to a full embedding using a slightly modified version of a depth first search. JGraphEd is able to store the order of the edges around each node through the use of the half edge data structure (see section 2.1.3).

The embed operation makes use of the same node and edge extenders used by the planarity testing operation.

4.2.9 Canonical Ordering

The CanonicalOrderOperation class provides methods for performing a canonical ordering of an embedded, maximal planar input graph. A canonical ordering of a maximal (triangulated) planar graph G embedded in the plane, with exterior nodes u, v, w is a labeling of all the nodes $v_1 = u$, $v_2 = v$, v_3, … v_{n-1}, $v_n = w$ of G such that the following requirements are met:

- The sub graph G_{k-1} of G induced by v_1, $v_2 = v$, v_3, …, v_{k-1} is biconnected, and the boundary of its exterior face is a cycle C_{k-1} containing the edge {u,v}.
- v_k is in the exterior face of G_{k-1}, and its neighbours in G_{k-1} form an (at least 2-element) subinterval of the path $C_{k-1} - \{u,v\}$ [FPP90].

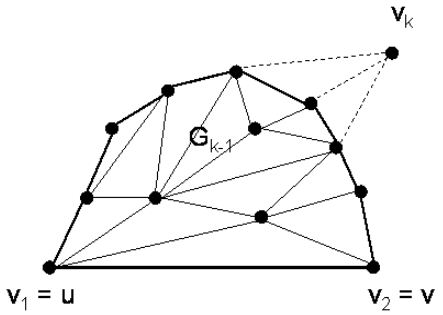

Figure 11 Illustration of the requirements of a Canonical Ordering

A simpler definition of a canonical ordering is that it is an ordering of the nodes of a graph such removing them in that (reversed) order will always leave behind a graph that is biconnected.

JGraphEd implements a linear time algorithm for computing the canonical ordering devised by the author. The steps are as follows:

- Make the graph maximal planar (and embed it)
- For each node of the input graph
  - Set the 'outer face edge count' to 0.
  - Set 'is on outer face' to false.
- Pick a random edge, and find the next/previous edges around the end nodes of this edge that have the same other end node and are adjacent to the same face. The 3 nodes of these 3 edges will form the outer triangle.
- Set the canonical label of the three nodes of the outer triangle to 1, 2, and N.
- For each of the three nodes of the outer triangle
  - Set the 'outer face edge count' to 2
  - Set 'is on outer face' to true
- For each of the neighbouring nodes of the three nodes of the outer triangle
  - Increment the 'outer face edge count' by 1

- Add the node with canonical label N to a doubly linked list of candidate nodes.
- For k = N down to 3
    - Pick any candidate node c from the candidate list.
    - Set the canonical label of c to k.
    - For every node v connected by an edge to the candidate node c
        - If 'is on outer face' of v is false
            - Set 'is on outer face' of v to true
            - For every node u connected by an edge to v
                - Increment the 'outer face edge count' of u by 1.
                - If u does not have canonical label 1 or 2
                    - If u has an 'outer face edge count' of 2, is not already in the candidates list, and 'is on outer face' is true, add it to the candidates list
                    - If u has an 'outer face edge count' not equal to 2, and is in the candidate list, remove it from the candidates list
        - Decrement the 'outer face edge count' of v by 1.
        - If v does not have canonical label 1 or 2
            - If v has an 'outer face edge count' of 2, is not already in the candidates list, and 'is on outer face' is true, add it to the candidates list
            - If v has an 'outer face edge count' not equal to 2, and is in the candidate list, remove it from the candidates list
        - Delete (flag for ignore) the edge between c and v

Every node is selected as a candidate exactly once, and becomes a part of the outer face exactly once. Each time either case occurs, the time spent is proportional the degree of the node. In summing over all nodes the algorithm runs in time proportional to the sum of the degrees of all nodes, which by planarity and Euler's formula will be linear in the number of nodes.

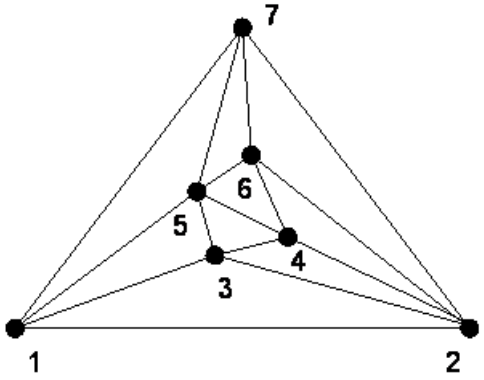

Figure 12 Illustration of a Canonical Ordered Graph

The canonical ordering operation requires the following additional fields for the nodes:

- Node – Integer, Canonical Number.
- Node – Integer, Outer Face Edge Count.
- Node – Boolean, Is On Outer Face.
- Node – Node, Previous Candidate in the List.
- Node – Node, Next Candidate in the List.

These fields are defined in the CanNodeEx and CanEdgeEx classes. The purpose of these fields should be clear from the description of the algorithm.

4.2.10 Normal Labeling

The NormalLabelOperation class provides methods for performing a normal labeling of maximal planar input graph. A normal labeling of a triangular Graph G is a labeling of all of the angles of its triangular faces such that the following conditions are satisfied:

a) For every triangle, there is one angle labeled with 1, a second labeled with 2 and a third labeled with 3.
b) The angles of every triangle in counter-clockwise order are 1, 2 and then 3.
c) For every interior node, the labels of the angles of its adjacent triangles consist of (non-empty) intervals of 1's followed by 2's and then 3's.
d) All of the angles at an exterior node have the same label.
e) The three exterior nodes whose angles are all 1, 2, 3, appear in counter-clockwise in this order around the exterior triangle.

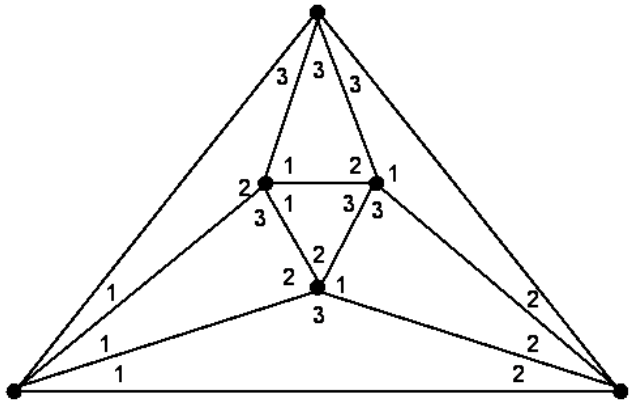

Figure 13 Illustration of a Normal Labeled Graph

Another property of a normal labeling is that for every edge between two adjacent internal triangles, the four labels at its ends consist of a pair of angles labeled i, j (i != j) at one end, and a pair of angles both labeled k (k != i, j) at the other. (See figure 15 for some examples). This fact is used to introduce a labeling of the interior edges of the

graph. Each interior edge is labeled with k, and directed towards the end where k appears twice.

The result of this edge labeling is three sets of edges with labels 1, 2 or 3 (referred to as T_1, T_2 and T_3 respectively). These three sets are referred to as a realizer of the triangular graph. The third property of the normal labeling implies several properties about a realizer:

i) Each interior node has exactly one departing edge labeled with k for each k in {1, 2, 3}.
ii) The counter-clockwise order of the edges incident to each node is:

1. Departing with label 1.
2. Arriving with label 3.
3. Departing with label 2.
4. Arriving with label 1.
5. Departing with label 3.
6. Arriving with label 2.

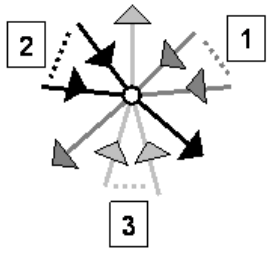

Figure 14 Illustration of the order of edges around a Normal Labeled Node

The fourth and fifth properties of normal labeling and the use of Euler's formula also tells us that none of the three exterior nodes have arriving edges, and all of their departing edges have the same label. Thus the exterior nodes with edges labeled k can be regarded as the root or sink of T_k.

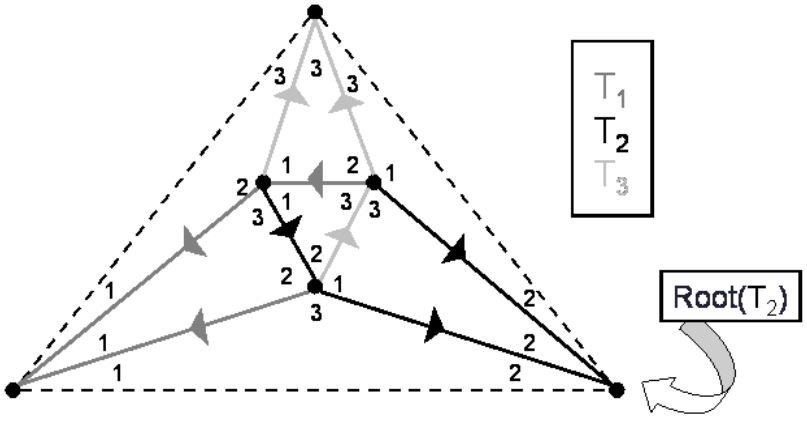

Figure 15 Illustration of the Realizers of a Graph

Construction of the normal labeling (and associated realizer) in JGraphEd is achieved through a series of edge contraction operations. An interior edge {x, y} in the triangular graph is contractible if x and y have exactly two common neighbouring nodes. The result of an edge contraction is that the edge {x, y} is removed and an interior edge {x, z} is added to every neighbour z of y that is not already a neighbour of x.

Since an edge contraction on a triangular graph always returns another triangular graph with one fewer node, it is possible to construct the normal labeling by reversing a sequence of edge contractions on the empty exterior triangle. A canonical ordering of the nodes (see section 4.2.9) of the graph allows each interior node to be added incrementally so that it is always initially connected to an exterior node. Interior nodes connected to an exterior node can always be correctly labeled since the interior edges at the exterior node must all have the same label and direction.

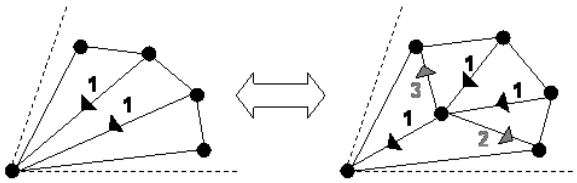

Figure 16 Illustration of the Edge Contraction Process

The normal labeling operation requires the following additional fields for the nodes and edges:

- Node – Node, Realizer 1 Parent.
- Node – Node, Realizer 2 Parent.
- Node – Node, Realizer 3 Parent.
- Node – Integer, Canonical Number.
- Edge – Integer, Normal Label Number.

These fields are defined in the NormalNodeEx and NormalEdgeEx classes. The three node fields are the references to a node's parent nodes in the 3 realizer trees formed by the normal labeling. The purpose of the remaining fields should be clear from the description of the algorithm.

4.2.11 Schnyder Straight Line Grid Embedding

The SchnyderEmbeddingOperation class provides methods for computing a straight line grid embedding of the embedded, maximal planar input graph with n nodes on an n-2 by n-2 grid. A straight line grid embedding of a graph is a drawing of the graph where all of the nodes occur at grid intersection points, and none of the edges are crossing.

JGraphEd implements the linear time algorithm for computing a straight line grid embedding proposed by Walter Schnyder in [Sch90]. The first phase of Schnyder's algorithm is the computation of a normal labeling of the input graph (see section 4.2.10). The second phase of Schnyder's algorithm is the determination of the grid coordinates for each interior node of the graph. This process relies on the normal labeling of the first phase.

Schnyder's algorithm uses the fact that a normal labeling uniquely partitions the input graph into 3 distinct regions for each node. These regions are bounded by the 3 unique paths from the node to each of the root nodes of the three realizers of the normal labeling.

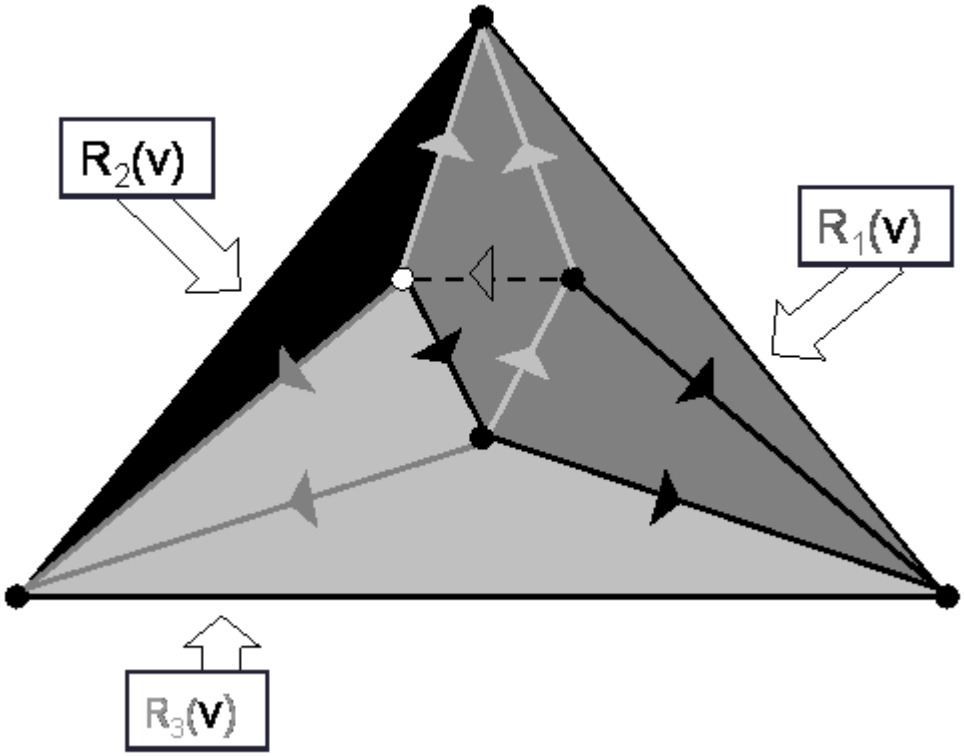

Figure 17 Illustration of the 3 Regions of a Node (shown in white)

The determination of the grid coordinates for Schnyder's algorithm requires calculating the following values for each node in the graph, for I = {1, 2, 3}:

- The I-Path count, which is the number of nodes along the path from the node to the root node of realizer I.
- The I-Sub-Tree count, which is the number of nodes in the sub tree of realizer I, rooted at the node.
- The I-Region count, which is the number of nodes in region I for the node.

All of these values may be calculated for each node by using a constant number of traversals of each of the realizer trees. JGraphEd uses 3 traversals of each of the 3 realizer trees, for a total of 9 traversals. Once these values have been computed, the grid coordinates of each node are trivial to compute. The X coordinate of each node can be set to its (I=1)-Region count minus its (I=3)-Path count. They Y coordinate of each node can be set to its (I=2)-Region count minus its (I=1)-Path count.

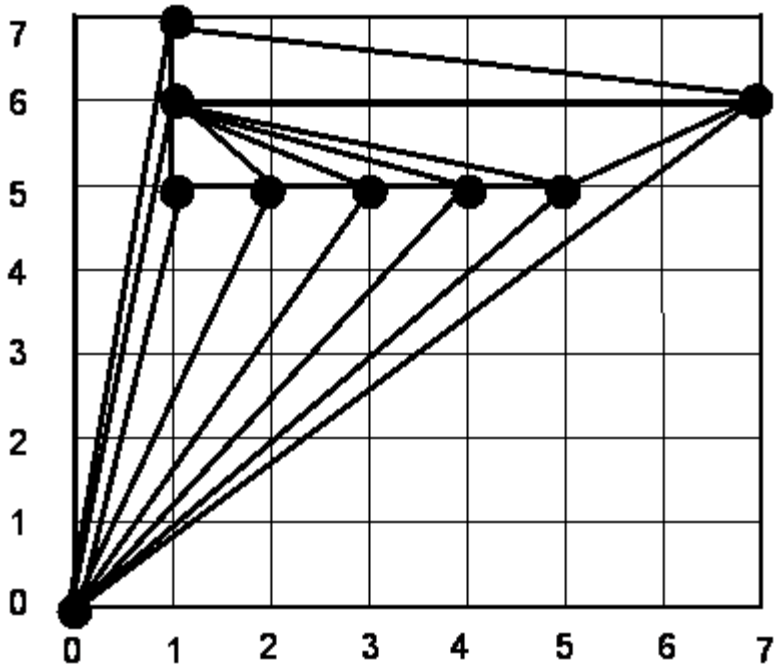

Figure 18 Illustrations of a Schnyder Straight Line Grid Embedding (9 nodes, 7x7 grid)

The Schnyder straight line grid embedding operation requires the following additional fields for the nodes and edges:

- Node – Integer, Canonical Number
- Node – Node, Realizer 1 Parent.
- Node – Node, Realizer 2 Parent.
- Node – Node, Realizer 3 Parent.
- Node – Integer (array of 3), I-Path Count for each Realizer.
- Node – Integer (array of 3), I-Sub-Tree count for each Realizer.
- Node – Integer (array of 3), I-Region count for each Realizer.
- Edge – Integer, Normal Label Number

These fields are defined in the SchnyderNodeEx and SchnyderEdgeEx classes. The purpose of these fields should be clear from the description of the algorithm.

4.2.12 Tree

The TreeOperation class provides two useful methods for determining whether an input graph has cycles, and determining whether or not an input graph is a binary tree.

The cycle testing method, which is used to determine whether or not the input graph is a tree, is implemented to run in linear time. This method first tests the planarity of the input graph (Section 4.2.7). If the graph is not planar, than clearly it must have

cycles. If the graph is planar, than a depth first search (Section 4.2.2) can be run on the graph in time linear to the number of nodes (by Euler's formula and Planarity). If the depth first search found any back edges, than the input graph had cycles, otherwise, the input graph had no cycles.

The binary tree testing method first uses the cycle testing method to determine whether or not the input graph is a tree. It then checks each node to ensure that it has at most 3 incident edges (2 for the root node). This method also runs in time linear to the number of nodes.

4.2.13 Chan Tree

The ChanTreeDrawOperation class provides three methods for invoking the three methods for drawing binary trees, proposed by Timothy Chan in [Cha99].

Chan's three methods for drawing binary trees all enforce the following criteria to the resulting drawings:

- The drawing must be **planar**; none of the edges of the drawing may cross each other.
- The drawing must be **strictly upward**; every node must be drawn below (in the Y coordinate axis) its parent in the tree.
- The drawing must be **strongly order preserving**; the edge to the left child of a node must have a smaller (or equal) X coordinate, and the edge to the right child of a node must have a larger (or equal) X coordinate.
- The drawing must be **straight-lined**; all edges from a parent node to its child node must be drawn as straight lines.

All of Chan's tree-drawing methods are also grid-drawings, in that every node is constrained to appear at the intersection of grid lines.

The first and second of Chan's binary tree drawing methods both draw the binary tree recursively in a bottom-up fashion. Each node in the tree is the root of a sub-tree which has an associated bounding box. Leaf nodes at the bottom of the tree have an empty bounding box. The key to both of these methods is determining how to combine or merge the bounding boxes of sub-trees associated with the child or children of internal (non-leaf) nodes. To answer this question, Chan proposes the use of a Left and Right rule for combining the bounding boxes of the left and right children of a node v.

In the left rule, the node v is vertically aligned with the right child, and the bounding box of the left child is placed one grid level below and one grid level to the left of v. The bounding box of the right child is then placed immediately below that of the left child. The right rule is defined symmetrically to the left rule.

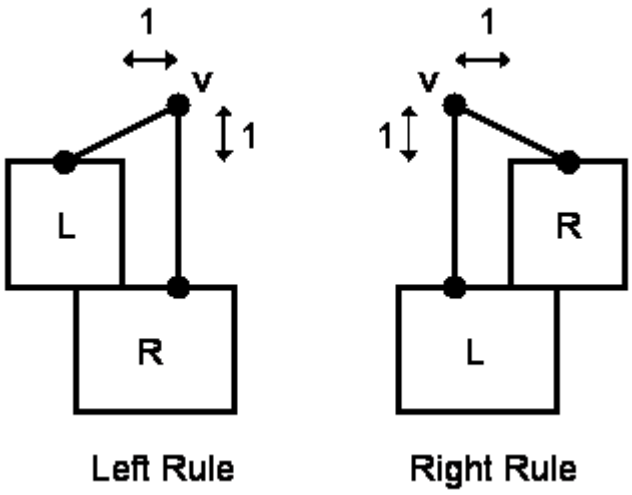

Figure 19 Illustrations of (regular) Left and Right Rules for Chan's Algorithm

The difference between the first two methods is in how they choose when to apply either the left or right rule.

For the first method, the left rule is used if the number of nodes in the sub-tree rooted at the left child is less than the number of nodes in the sub-tree rooted at the right child. Otherwise, the right rule is used. This method results in a drawing of a binary tree with n nodes on a grid with O(n) height, and $O(n^{0.695})$ width.

For the second method, the size (number of nodes) of the biggest left and right sub-tree seen thus far by nodes along the recursive path is maintained. The left rule is used if the number of nodes in the sub-tree rooted at the left child + the biggest right sub-tree is less than the number of nodes in the sub-tree rooted at the right child + the biggest left sub-tree. Otherwise, the right rule is used. This method results in a drawing of a binary tree with n nodes on a grid with O(n) height and $O(\sqrt{n})$ width.

The third method also draws binary trees recursively in a bottom-up fashion. The main difference between this method and the first two methods is that it uses extended left and right rules in addition to the ‘regular’ left and right rules to combine the bounding box of a node’s children. The extended left rule differs from the ‘regular’ left rule in that the bounding box of the right child may be shifted in the positive X coordinate direction. The extended right rule is defined symmetrically.

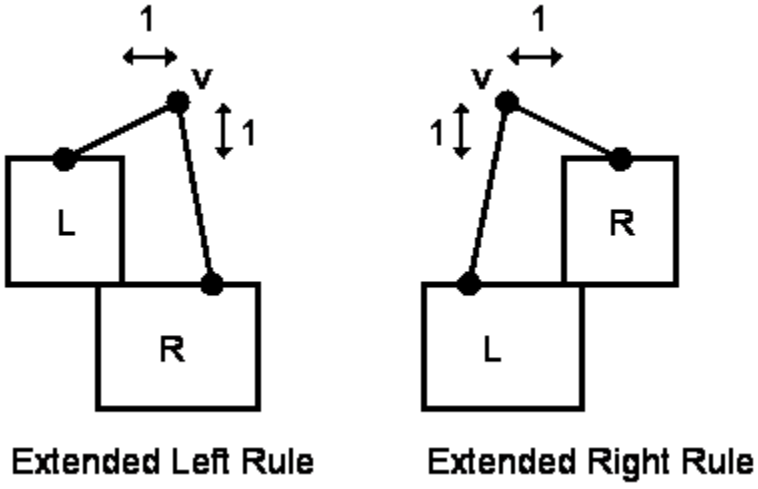

Figure 20 Illustrations of Extended Left and Right Rules for Chan's Algorithm

The first phase of the third method requires computing the index k of the last node whose sub-tree size is greater than $n - (n / 2^{\sqrt{(2\log n)}})$. In the second phase, bounding boxes are merged recursively. For nodes with indices less than k-2, the procedure for method 1 is applied. At the node with index k-1, the extended right rule is applied if the node with index k is a left child, and the extended left rule is applied otherwise. When an extended rule is applied, the bounding box of the appropriate child is shifted into alignment with the left-most or right-most boundary box edge of its ancestors in the tree. For nodes with indices greater than k-1, the right rule is applied if the node with index k was a left child, and the left rule is applied otherwise. This method results in a drawing of a binary tree with n nodes on a grid with height at most n-1, and $O(2^{\sqrt{(2\log n)}} \sqrt{\log n})$ width.

It is interesting to note that there appears to be a typographic error in Chan's paper. In the description of the third method, it is written that the extended and 'regular' LEFT rules should be applied if the node with index k is a left child, yet the subsequent figure clearly shows that the RIGHT rules should be used in this case.

It is worth noting that JGraphEd implements a third 'regular' rule and extended rule (called 'other rule' in the source code) to handle the appropriate cases where a node being processed has only one child.

In order to implement Chan's three methods for drawing binary trees in linear time, JGraphEd performs a traversal of the tree to compute the size of each sub-tree prior to executing any of the 3 methods. Also, instead of computing absolute grid coordinates for the bounding boxes of children of a node during the recursion, JGraphEd maintains fields at each node that track how much it has been shifted by in the X and Y directions. The width and height of the bounding box of sub-trees rooted at nodes does need to be computed during the recursion process. Because of this fact, each node also has fields to store its position to the bounding box of its sub-tree. At the end of the recursion process,

JGraphEd performs a traversal of the binary tree, adjusting each node's coordinates using the cumulative x and y shifts of its ancestors.

The Chan tree drawing operation requires the following additional fields for each of the nodes:
Node – Node, Tree Parent
Node – Integer, Sub-Tree size
Node – Integer, Grid Shift X
Node – Integer, Grid Shift Y
Node – Integer, Bounding Box Relative X
Node – Integer, Bounding Box Relative Y
Node – Integer, Bounding Box Width
Node – Integer, Bounding Box Height

These fields are defined in the ChanNodeEx and ChanEdgeEx classes. The purpose of these fields should be clear from the description of the algorithm. The Bounding Box Relative Y field could be removed since its value can never be anything except 0.

4.2.14 Dijkstra's Shortest Path

The DikstraShortestPathOperation class provides a method for computing the shortest path between any two nodes in the graph using Dijkstra's single-source shortest path algorithm [CLR90].

Dijkstra's shortest path essentially works by expanding a 'cloud' that eventually envelopes all of the nodes in the graph. The shortest path length of any node is known as soon as it is enveloped by the cloud. At each stage of the algorithm, the next node to be enveloped (and has not already been enveloped) is the one currently with the smallest shortest path length to the source. When a new node is enveloped, each of its neighbouring (connected by and edge) nodes has their distance to the source node decremented if they are closer to it via the newly enveloped node.

The implementation of Dijkstra's shortest path algorithm in JGraphEd prompts the user for a source and destination node. If a shortest path exists between these nodes, it is drawn to the graph by tracing back the edges used to reach the destination from the source. If no path exists, an appropriate message dialog is displayed to the user. JGraphEd's implementation makes uses of the extract min operation of the binary heap data structure (Section 3.3) to speed up the retrieval of the node with the smallest shortest path length to the source. The decrease key operation of this data structure is used to decrease the shortest path length of nodes when necessary. The use of the binary tree data structure allows this algorithm to run in O(mlogn) time for a node with n nodes and m edges.

The Dijkstra's shortest path operation requires the following additional fields for each of the nodes:

- Node – Real-Valued Number, Shortest Path length
- Node – Boolean, Is Enveloped
- Node – Edge, Trace back edge
- Node – BinaryHeapNode, Binary Heap Node

These fields are defined in the SPNodeEx and SPEdgeEx classes. The purpose of these fields should be clear from the description of the algorithm.

4.2.15 Minimal Spanning Tree

The MinimumSpanningTreeOperation class provides a method for computing the minimum spanning tree of the input graph. A minimum spanning tree is a tree (without cycles) that connects all of the nodes of the graph, and has the shortest total edge length (summed over all edges in the tree) possible.

JGraphEd implements Prim's algorithm for computing a minimum spanning tree [CLR90], which operates in a very similar fashion to Dijkstra's shortest path algorithm (Section 4.2.14). In brief, Prim's algorithm works by repeatedly selecting a node which has not yet been added to the tree that has an edge with the shortest length to a node that has already been added to the tree. Both the selected node, and associated shortest edge are added to the minimum spanning tree.

The implementation of Prim's algorithm in JGraphEd uses a binary heap data structure (Section 3.3) to store the nodes which have not yet been added to the minimum spanning tree. The key used to determine the position of these nodes within the heap is length of their shortest edge to any node that has already been added to the spanning tree. The decrease key operation of this data structure is used to decrease the shortest edge length of nodes when necessary.

The minimum spanning tree operation requires the following additional fields for each of the nodes:

- Node – Edge, shortest edge.
- Node – Real Valued Number, shortest edge length.
- Node – Boolean, Is Added To Tree.
- Node – BinaryHeapNode, Binary Heap Node.

These fields are defined in the MSTNodeEx and MSTEdgeEx classes. The purpose of these fields should be clear from the description of the algorithm.

5 Conclusion

This document has described all of the design aspects and features of JGraphEd. Section 2 described the framework for its editing and drawing capabilities. Section 3 described the data structures that were used both by the framework and by various graph algorithms or operations. Section 4 described the implementations of algorithms or operations which are provided with JGraphEd.

Hopefully, the reader has been left with a good understanding of how all of the different components interact with each other to work as a whole.

JGraphEd is released to the public as Open Source software, and the reader is free to modify or extend it in any way. The only restriction is that they also provide the source code of their modifications to the public.

Hopefully JGraphEd can serve some small part in educating students about graphs in general, and graph algorithms in particular.

6 References

[BSK+00] M. de Berg, O. Schwarzkopf, M. van Kreveld, M. Overmars,
Computational Geometry: Algorithms and Applications,
Springer-Verlag, Second Edition, 2000.

[BL76] K.S. Booth and G.S. Lueker,
Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using PQ-Tree Algorithms,
Journal of Computer and System Sciences, 13, pp. 335-379, 1976.

[Bou96] P. Bourke
Bézier curves, 1996
http://astronomy.swin.edu.au/~pbourke/curves/bezier/

[BMR+00] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal,
Pattern-Oriented Software Architecture, Volume 1: A System of Patterns, 2nd edition,
John Wiley & Sons Ltd., 2000.

[Cha99] T. M. Chan,
A Near-Linear Area Bound for Drawing Binary Trees
In *Proc. 10th Annual ACM-SIAM Sympos. on Discrete Algorithms*, pp. 161-168, 1999.

[CNA+85] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa,
A linear algorithm for embedding planar graphs using PQ-trees,
Journal of Computer and System Sciences, 30(1), pp. 54-76, 1985.

[CLR90] T. H. Cormen, C. E. Leiserson, R. L. Rivest,
Introduction to Algorithms, MIT Press, Cambridge, Massachusetts, 1990.

[Dav00] M. Davidson
Using the Swing Action Architecture, 2000
http://java.sun.com/products/jfc/tsc/articles/actions/

[Eve79] S. Even
Graph Algorithms, Computer Science Press, Rockville, Maryland, 1979.

[FPP90] H. de Fraysseix, J. Pach and R. Pollack,
How to draw a planar graph on a grid,
Combinatorica, 10, pp. 41-51, 1990.

[GHJ+94] E. Gamma, R. Helm, R. Johnson, J. Vlissides,
Design Patterns, Elements of Reusable Object-Oriented Software,
Addison-Wesley, 1994.

[Sch90] W. Schnyder,
Embedding Planar Graphs on the grid,
In *Proc. 1st ACM-SIAM Symposium on Discrete Algorithms*, pp. 138-148, 1990.

[Unk04] Uknown Author,
How to use Internal Frames,
http://java.sun.com/docs/books/tutorial/uiswing/components/internalframe.html, 2004.