

Design of a Graph Drawing and Visualization System

Ragib Hasan, Meetesh Barua, Pradipta Prometheus Mitra, Jalaluddin Mahmud,
Md. Saidur Rahman*

Department of Computer Science & Engineering,
Bangladesh University of Engineering & Technology
Dhaka-1000, Bangladesh

*Nishizeki Laboratory
Graduate School of Information Sciences
Tohoku University, Sendai, Japan

Emails: ragibhasan@yahoo.com, meeteshbarua@yahoo.com, shmitra@yahoo.com, mrony@accesstel.net,
saidur@nishizeki.ecei.tohoku.ac.jp

Abstract: *The visualization of complex conceptual structures is a key component of support tools for many applications in science and engineering. A graph is an abstract data structure that is used to model information. Hence, many information visualization systems require graphs to be drawn so that they are easy to read and understand. Also many practical applications like VLSI design need the drawing of a graph in some specific way. In this paper, we address the problem of drawing and visualization of graphs using various algorithms by presenting an integrated software environment for both the design and the visualization of graphs.*

Keywords: *Graph drawing, Orthogonal drawing, st-graph, Constrained visibility, Java.*

1. INTRODUCTION

In recent years graph drawings have appeared as a lively area in computer science, and many algorithms have emerged to produce graphs in different styles [1]. However, only a few algorithms are published with implementation, although the interest in experimentally testing the performance of graph drawing algorithms has increased in recent years. To bridge this gap, our attempt is to design an integrated platform of graph drawing where these algorithms can be easily implemented and tested. Our tool, BUET Visual Graph Toolkit is generalized software for drawing and visualization of graphs. This software is aimed at creating an easy-to-use and integrated system of tools for the drawing of various types of graphs that are frequently used in Computer Science and in Technical fields.

The Software (referenced as VisiGraph from now on) combines an integrated graph designer (gDesigner) which can be used to draw various types of graphs using a point-and-click mouse interface. The graph can then be visualized using various graph drawing algorithms. The main goals in designing the software were:

- i. A simple, graphical user interface for design and visualization of graphs.
- ii. Point and click input interface using a mouse for drawing the graph.
- iii. A generalized framework for implementation of any type of graph drawing algorithm.
- iv. Separation of designing, drawing and viewing components to facilitate robustness and flexibility.
- v. Use of the Java™ programming language in order to achieve cross-platform coverage [4].
- vi. Modular Object oriented design to tackle the complexity of code management and implementation.

2. OVERVIEW

This section shall provide a general overview of the main modules of the software developed.

The main components of this software system are:

- i. **Central dispatcher**
This is a visual interface from which the drawing tool and the algorithm components are invoked.
- ii. **A graph drawing tool**
The main graph drawing tool **gDesigner** is a generalized tool for drawing of graph images. It implements the design philosophy of simplicity by using a point and click mouse interface for drawing of images.
- iii. **A collection of visualization algorithms**
This collection emphasizes not only the implementation of visualization algorithms but also generates actual layouts in user-friendly modes. Other than the visual representation presented on the screen, the software is capable of printing and generating postscript output. These algorithms and their implementation issues will be discussed in the subsequent sections.

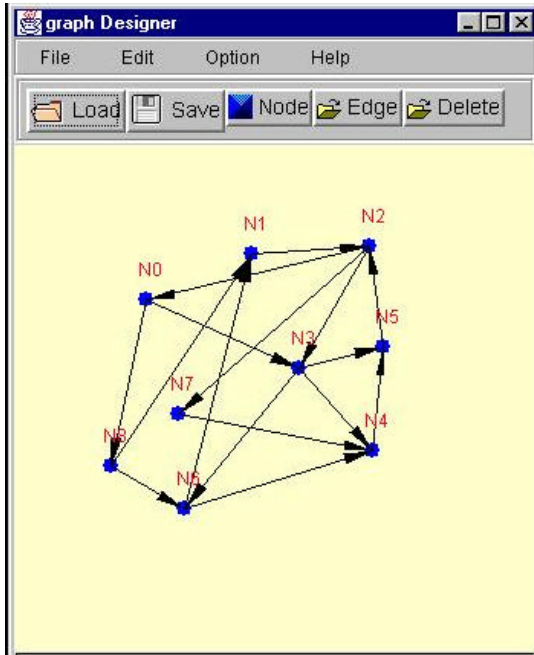


Fig 1: gDesigner

3. HV-TREE DRAWING

In this section, the implementation of the first set of algorithms, namely HV-TREE Drawing, is presented.

HV-drawing (“HV” stands for horizontal-vertical) of a binary tree can be defined as a straight line drawing such that ,for every vertex u :

- i. A child of u is either horizontally aligned with and to the right of u , or vertically aligned with and below u
- ii. The bounding rectangles (smallest rectangle with horizontal and vertical sides covering the drawings) of the sub-trees of u do not intersect.

3.1 Types of HV- Tree Drawing

HV-Drawing is planar, straight-line, orthogonal and downward. It can be of 3 main types

- i. Horizontal Combination
- ii. Vertical Combination
- iii. Horizontal-Vertical Combination

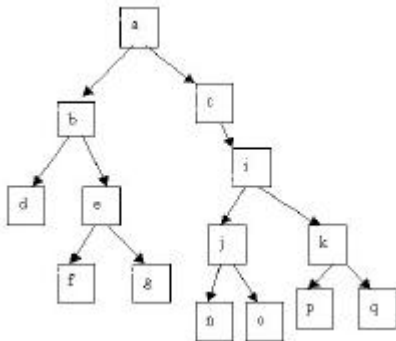


Fig (2.a) : A binary tree

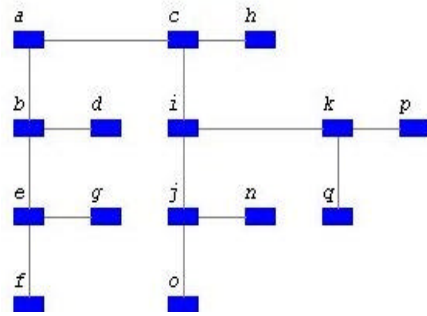


Fig (2.b) Horizontal-vertical combination of the tree shown in Fig(2.a)

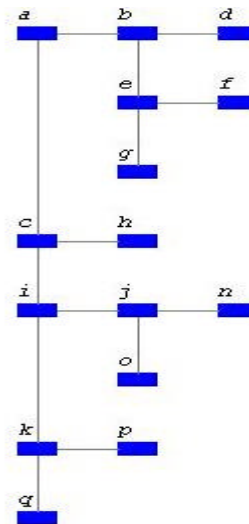
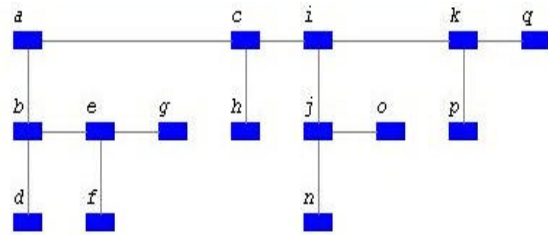


Fig (2.c) Horizontal and Vertical Combinations

3.1.2 Implementing HV- Tree Drawing

A general divide-and-conquer scheme for constructing hv-drawings works as follows

Divide : Recursively construct hv-drawings for the left and right sub-trees.

Conquer: Perform either a horizontal combination or a vertical combination.

In our case the sub-tree with the largest number of vertices was placed to the right of the other one (in case of horizontal combination), below (in case of vertical combination) and alternately right or down (in case of horizontal-vertical combination)

3.2 Implementing Visual drawings for a planar st – graph

The system implements several visual drawing approaches for a planar st- graph. A planar st graph is an acyclic digraph which has a single source and a single sink.

3.2.1 Definition of the representations

The implemented visual algorithms are visibility representation, constrained visibility representation and orthogonal from visibility representation.

3.2.1.1 Visibility Representation : Let G be a planar st- graph. A visibility representation P of G draws each vertex v as a horizontal segment, called vertex segment $P(v)$ and each edge (u, v) as a vertical segment, called edge segment $P(u, v)$, such that

- i. The vertex segments do not overlap.
- ii. The edge segments do not overlap.
- iii. Edge segment $P(u, v)$ has its bottom endpoint on $P(u)$, its top endpoint on $P(v)$ and does not intersect any other vertex segment.

3.2.1.2 Constrained Visibility Representation : A Constrained visibility constructs a visibility representation of a planar st graph, such that some pre specified edges are vertically aligned.

3.2.1.3 Orthogonal From Visibility : Orthogonal from visibility constructs an orthogonal drawing from the visibility representation of the planar st graph.

3.2.2 Systematic implementation of Visibility representation

3.2.2.1 Algorithm :

A fast polynomial time algorithm for visibility representation [1] is used for the efficient implementation.

3.2.2.2 Implementation :

To implement the algorithm it is necessary to obtain a digraph G^* from the given input st- graph G . The vertex set of G^* is the set F of faces of G (including the external faces). To construct G^* , the faces (both internal and external) for the digraph G must be computed in a depth first search approach with circular link list representation of the digraph.

When the digraph G^* is computed from the given input graph, next step is to calculate the optimal weighted topological numbering for the digraph G and G^* . An optimal weighted topological numbering can be obtained by assigning to each vertex a number equal to the number of edges on a longest directed path terminating at that vertex.

Next, using the topological numbering, horizontal and vertical line segments are drawn. horizontal line segments are drawn for each vertex v of the digraph G . The horizontal line segments are drawn in such a way that the y coordinate of the line segments are topological numbering corresponding to that vertices. The left x coordinate for vertex v is the topological numbering for the left(v) and the right x coordinate for vertex v is the topological numbering for the right(v)-1. Vertical line

segments are drawn for each edge e of the digraph G . The vertical line segments are drawn in such a way that the x coordinate of the line segments are topological numbering corresponding to left face of that edge. The bottom y coordinate for edge e is the topological numbering for the orig(e) and the top y coordinate for edge e is the topological numbering for the dest(e).

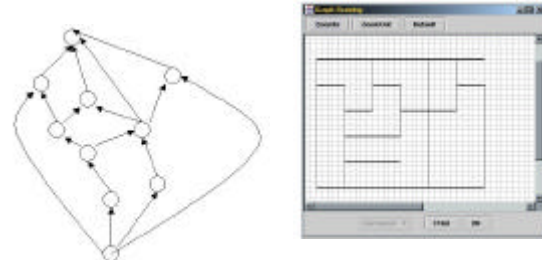


Fig 3 : A planar st- graph and its visibility representation .

3.2.3 Constrained Visibility

3.2.3.1 Algorithm : A fast polynomial time algorithm presented in [1] for constrained visibility representation is used.

3.2.3.2 Implementation : In the input a collection of non intersecting paths of G is given as path constrained. The algorithm constructs the visibility representation for each path. In order to implement the algorithm the first step is to calculate another digraph called $G[\Pi]$ from the input graph. An optimal weighted topological numbering for both the digraph G and $G[\Pi]$ has to be computed with the exception that the source vertex for the digraph $G[\Pi]$ is assigned to topological number $-1/2$ but the source vertex for the digraph G is assigned to topological number 0. Vertical line segments are drawn for each edge e of each path Π . The vertical line segments are drawn in such a way that the x coordinate of the line segments are topological numbering corresponding to that path. The bottom y coordinate for edge e is the topological numbering for the orig(e) and the top y coordinate for edge e is the topological numbering for the dest(e). Horizontal line segments are drawn for each vertex v of the digraph G . The horizontal line segments are drawn in such a way that the y coordinate of the line segments are topological numbering corresponding to that vertex v . The left x coordinate for vertex v is the minimum topological numbering of the path where the vertex is present. The right coordinate for vertex v is the maximum topological numbering of the path where the vertex is present.

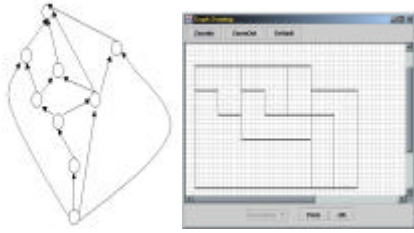


Fig 4 : A planar st- graph and its constrained visibility representation .

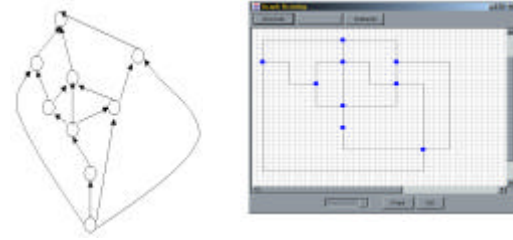


Fig 5 : A planar st- graph and its orthogonal representation .

3.2.4 Orthogonal from Visibility

3.2.4.1 Algorithm :

A fast polynomial time algorithm as in [1] for orthogonal from visibility representation is described below:

- i. Construct a planar embedding of G and orient its edges such that the resulting digraph D is a planar st-graph.
- ii. Create a set of $n-2$ directed paths of D associated with the vertices of D distinct from s and t . Unify paths sharing edges, which yield a set Π of nonintersecting paths.
- iii. Using the algorithm of constrained visibility, construct a constrained visibility representation F of D , with respect to the set Π of paths, such that F has integer coordinates.
- iv. Construct a planar orthogonal grid drawing of G as follows:
 - a) For each vertex v distinct from s and t , draw v at the intersection $P(v)$ of vertex segment $F(v)$, with edge segments of path Π .
 - b) Draw vertex s (resp. t) at the intersection of its vertex segment with the edge segment of its median outgoing (resp. incoming) edge.
 - c) For each edge $e=(u, v)$ such that u and v are distinct from s and t , draw e as an orthogonal chain through the following points : $P(u)$, the intersection of $F(u)$ and $F(e)$, the intersection of $F(e)$ and $F(v)$, and $P(v)$. The chain consists of three segments, where the first and last segment may be empty.

3.2.4.2 Implementation : First step of the algorithm is not necessary if the given graph is a planar st- graph. So the input graph is searched to find $n-2$ directed paths. These paths act as constraints for calculating the constraint visibility representation. Intersections of vertex segment with path segment are also calculated to draw the orthogonal chain. Orthogonal chains are drawn according to the rules of the algorithm.

Applications of HV-Tree Drawing

- i. Lisp programs can be visualized with hv-drawing.
- ii. It has application in drawing of trees with better aspect ratio.

4. DFS HEURISTIC FOR ORTHOGONAL GRAPH DRAWING

The DFS heuristic implements an algorithm for the orthogonal drawing of graphs, which is based on an algorithm, presented in [3]. This section describes the implementation issues and output of the software.

4.1 Algorithms used in the Implementation

Numerous algorithms of various kinds were implemented during the implementation of the software. Here, we present the main algorithms used

a) DFS

The Depth first search is a classic algorithm of Graph Theory and can be found in many standard computer science texts, such as [4].

b) Node Placement

Algorithm Node_placement

- i. pick an arbitrary node v_1 and perform $DFS(v_1)$
- ii. Process each node in DFS order
- iii. for each node to be processed
- iv. the first node is placed in the horizontal and vertical slot.
- v. give the node either a RED or BLUE placement

The above algorithm requires two definitions for clarification:

RED Placement: This placement consists of adding a new vertical slot at the extreme right of the current drawing and placing the node in the horizontal slot of its parent.

BLUE Placement: This placement consists of adding a new vertical slot at the extreme bottom of the current drawing and placing the node in the vertical slot of its parent.

c) Edge Routing

Algorithm Edge_Routing

- i. For each forward edge
- ii. If it is a red edge

- iii. first assign the edge along a horizontal segment and then along a vertical segment
- iv. else
- v. first assign the edge along a vertical segment and then along a horizontal segment

d) Port Assignment

The most important of all algorithms used. We present the algorithm here. For more details with theorems and explanations, one can refer to pages 10-12 of [3].

- i. for $i = 1 \dots k$ do
- ii. if $i = 1$ then
- iii. Let e_1, e_2, \dots, e_l be the incoming blue non-tree-edges of w_l .
- iv. Sort e_1, e_2, \dots, e_l by decreasing column of the bend of e_j .
- v. In case of a tie, sort by decreasing row of endpoint $!= w_j$ of e_j .
- vi. Set $r_p = -1$;
- vii. for $j = 1, \dots, l$ do
- viii. assign the horizontal segment of e_j to row $r_p - j + 1$
- ix. Set $R = r_p - 1 + l$;
- x. else
- xi. Set $l = 0$
- xii. Let r_p be the row of the incoming tree-edge of w_i .
- xiii. Let e_1', \dots, e_r' be the outgoing red edges of w_i
- xiv. Sort e_1', \dots, e_r' by decreasing column of the endpoint $!= w_i$ of e_j' .
- xv. for $j = 1, \dots, r$ do
- xvi. assign the horizontal segment of e_j to row $r_p - j + 1$
- xvii. Set $R = \min\{R, r_p - r + 1\}$
- xviii. Extend the box of w_i to cover the rows $r_p, \dots, r_p - \max\{l-1, r-1, 0\}$

4.2 Design & Architecture

We decided to completely separate the detection and visualization parts of the orthogonal drawing. The detection part was designed as follows:

4.2.1 Data Structures

- i. **EdgeClass:** Class representing an edge. One incident node is implicit, the other is stored. Information is stored about the mirror edges and edge routing co-ordinates.
- ii. **Node:** Contains the Id, edgelist(a linked list of edgeclass), co-ordinates and related information.

4.2.2 Algorithm Implementation

The aforementioned classes are the core data structures. They reflect none of the algorithmic side of the implementation. Three main **Graph Classes** to that part of the job. These classes are:

- i. **GenericGraph:** The name of the class expresses what it is all about. It is a very generic class that only establishes the data-structures by reading the

information in from a file and computes the DFS tree of the edge. The DFS index of nodes is an essential information required in almost any graph drawing algorithm. This class can be used as a starting point for any graph drawing algorithm. The main methods are *createGraph*, *dfs* and *dfsVisit*.

- ii. **GraphNPlacemntEroute:** An class inherited from **GenericGraph**. The main methods are *nodePlaceMent* and *edgeRouting*. They perform the all important job of node placement(the choice of red or blue placement is done by a random number) and that of edge routing.
- iii. **GraphPAssign:** Inherited from **GraphNPlacemntEroute** and hence inherits all the attributes from the former two graph classes. It completes the implementation of the algorithm by three classes:

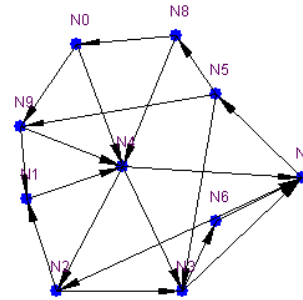


Fig 6(a): A graph designed by gDesigner

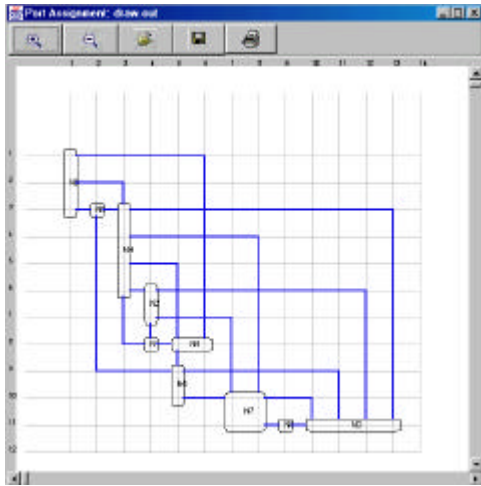


Fig 6(b): The corresponding Orthogonal Drawing

REFERENCES

- [1] G. D Battista., P. Eades, R. Tamassia, I. G. Tollis: *Graph drawing: Algorithms for the Visualization of Graphs*, Prentice Hall Inc., Upper Saddle River, New Jersey, 1999.
- [2] T. H. Cormen, C.E. Leiserson, R.L. Rivest, : *Introduction to Algorithms*, MIT Press, 1990.
- [3] T. Biedl: *The DFS-Heuristic for orthogonal graph drawing*, *Computational Geometry: Theory and Applications*, 18, 2001, pp. 167-188.
- [4] E. Horowitz, S. Sahni: *Fundamentals of Computer Algorithms*.
- [5] Deitel & Deitel: *Java 2: How to program*, Prentice Hall Inc., Upper Saddle River, New Jersey, 2000.
- [6] T. Calamoneri, S. Jannelli, R. Petreschi, *Experimental comparison of graph drawing algorithms for cubic graphs*, *Journal of Graph Algorithms and Applications*, 3(2), 1999, pp. 1-23.