A Heuristic for the Constrained One-Sided Two-Layered Crossing Reduction Problem
for Dynamic Graph Layout

by

Dung Mai

A dissertation proposal submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Graduate School of Computer and Information Sciences
Nova Southeastern University

2009

An Abstract of a Dissertation Submitted to Nova Southeastern University
in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

A Heuristic for the Constrained One-Sided Two-Layered Crossing Reduction Problem for Dynamic Graph Layout

by
Dung Mai

May 2009

Data in real-world graph drawing applications often change frequently but incrementally. Any drastic change in the graph layout could disrupt a user's "mental map." Furthermore, real-world applications like enterprise process or e-commerce graphing, where data increase rapidly, demand a comprehensive responsiveness when rendering the graph layout in a multi-user environment in real time. Most standard static graph drawing algorithms apply global changes and redraw the entire graph layout whenever the data change. The new layout may be very different from the previous layout and the time taken to redraw the entire graph degrades quickly as the amount of graph data grows. Dynamic behavior and the quantity of data of real-world applications pose challenges for existing graph drawing algorithms in terms of incremental stability and scalability.

A constrained hierarchical graph drawing framework and modified Sugiyama heuristic are proposed in this research. The goal of this research is to improve the performance and scalability of the constrained graph drawing framework while preserving layout stability. The primary innovation of the proposed constrained hierarchical graph drawing framework is to use a relational data model to improve the performance of the algorithm by reusing vertex and edge information stored in a relational database.

This research is based on the work of North and Woodhull (2001) and the constrained crossing reduction problem proposed by Forster (2004). The result of the proposed constrained hierarchical graph drawing framework and the new Sugiyama heuristic, especially the modified barycenter algorithms, will be tested and evaluated against North and Woodhull's (2001) online graph drawing framework.

Table of Contents

List of Tables

Table

List of Figures

Figure

Chapter 1

Introduction

General hierarchical graph layouts as shown in Figure 1 are often used to display

relationships between objects (North, 1995). Some examples of their use include entity

relationship models in databases, the Unified Modeling Language (UML) in software

engineering, management organizational charts, and hierarchical layouts in computer

networking to display Internet networks (Battista, Eades, Tamassia, & Tollis, 1999;

Cohen, Battista, Tamassia, Tollis, & Bertolazzi, 1992; North & Woodhull, 2001).
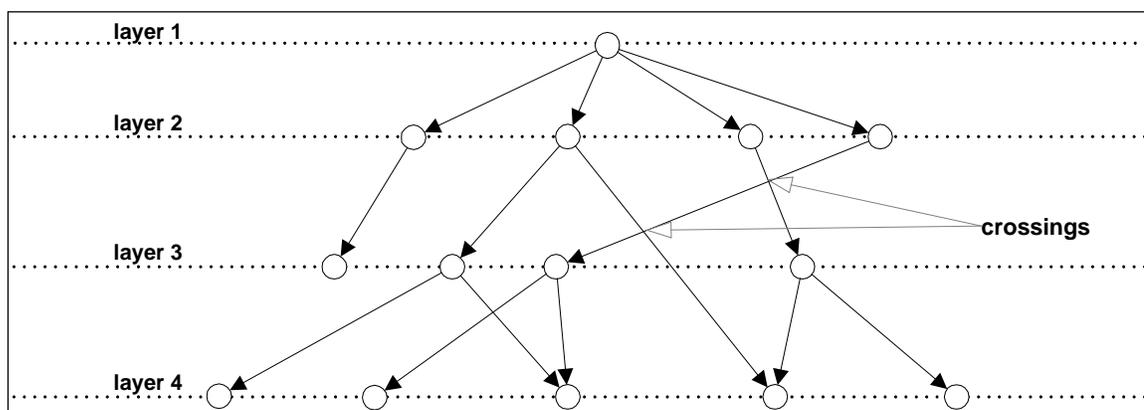
Figure 1. A hierarchical graph layout

Although current hierarchical graph layout algorithms have been well studied (North &

Woodhull, 2001) and have been effective for drawing static graphs with fewer than 100

nodes, real-world graph editing applications such as enterprise process modeling

applications depict large amounts of complex data that change frequently but

incrementally. Modelers who manage such large and complex models often

incrementally update the model locally and expect a corresponding local change in the

layout of the model without too drastic a change from the previous layout. Without this

precaution users could be confused and unable to associate the new model with the previous model. In other words, the users' "mental map," their perception of the graph's concepts based on previous knowledge, could be destroyed if the new layout is substantially different from the previous layout (Eades & Kelly, 1984). Thus, incremental stability is an important requirement of real-world graph applications. Currently, most standard graph drawing algorithms tend to apply global optimization and redraw the entire graph when the graph data change. The resulting layout can sometimes be quite different from the previous layout. Moreover, graph models of real-world applications are often updated by some users while others are viewing the model online. To support both editing and viewing in real-time mode, graph editing applications need to not only generate incrementally stable layouts but generate them as quickly as possible. Applying a global change on the model when a local change is made may not be scalable for large data models such as enterprise process models. As a result, the dynamic behavior of real-world graph applications poses challenges for current graph drawing algorithms in terms of incremental stability and scalability.

To address dynamic behaviors of real-world graph applications such as incremental stability and scalability, Cohen et al. (1992) proposed dynamic graph algorithms for different types of graph drawing techniques, especially series-parallel directed graphs and trees. Miriyala and Tamassia (1993) introduced an incremental graph drawing algorithm for drawing orthogonal graph layouts. Brandes and Wagner (1997) formulated dynamic layouts in terms of *random fields* and presented a formal concept for dynamic graph layouts. He and Marriott (1998) proposed several models for constrained

force-directed graph layout. Diehl, Görg, and Kerren (2000) presented an *off-line* dynamic graph drawing framework called *foresighted layout* that renders a sequence of graph layouts from a given sequence of graphs while preserving the global layout. Lee, Lin, and Yen (2006) presented a modified simulated annealing algorithm that preserves the user's mental map and ensures graph layout stability. Frishman and Tal (2007) proposed a new online dynamic graph drawing that is based on a force-directed layout algorithm. Examples of progress in hierarchical drawing for directed graphs in recent years include a hierarchical directed graph drawing system called *online dynamic graph drawing* (North, 1995; North & Woodhull, 2001), and hierarchical graph views for dynamic graph layouts (Buchsbaum & Westbrook, 2000; Raitner, 2004).

This section first discussed the dynamic nature of graph data in real-world applications in which data change frequently but incrementally. Any drastic changes in graph layouts could disrupt a user's mental map. It then addressed the limitation of current static graph drawing algorithms, which redraw the entire graph layout without taking into account the user's mental map. The section also introduced incremental graph layout algorithms as a solution to this problem. The next section will discuss the impacts and potential complexity of incremental graph drawing algorithms; especially, crossing reduction for one-sided two-layered graphs.

Problem Statement

Incremental graph drawing algorithms such as online dynamic graph drawing (North & Woodhull, 2001) for drawing hierarchical graph layouts do improve layout stability, but their aesthetic criteria impact the readability criterion by complicating the

computation of vertex reordering on a layer by imposing certain constraints on vertices. In fact, the crossing reduction problem for dynamic hierarchical graph drawing is a variant of the crossing reduction problem called *constrained crossing reduction,* in which the crossing reduction algorithm works with a set of predefined constraints. This problem is presented in Chapter 2.

Minimizing the number of crossings is a major goal for measuring the quality of a graph layout algorithm, but unfortunately crossing reduction is an NP-complete problem (Garey & Johnson, 1983). Though there has been significant research in this area, most of the work has focused on developing heuristics to solve the crossing reduction problem without constraints on vertices. Moreover, the recent experiment done by Huang and Eades (2005) showed that in some cases a constrained graph layout that produces more edge crossings provides a better visualization than the same layout with fewer edge crossings. User constraints indeed impact the readability of the drawing and influence the design of the algorithm that solves the crossing reduction problem. Hence, minimizing the number of crossings on a constrained graph layout requires further investigation.

Although several researchers propose efficient online dynamic graph drawing systems such as the online hierarchical graph drawing framework (North & Woodhull, 2001), those systems do not address the constrained crossing reduction problem. Further research is needed to address this problem, to improve the scalability and performance of the dynamic graph drawing algorithm, especially the crossing reduction problem as suggested by North and Woodhull (2001), and to utilize a relational database to store dynamic graph layouts. The work in the proposed dissertation will develop an

incremental graph drawing framework based on online dynamic graph drawing (North, 1995; North & Woodhull, 2001). The result of this research will be to improve the incremental stability and scalability of graph drawing systems by using relational data models to capture incremental graph layouts.

Goal

The objective of this research is (1) to develop a constrained graph drawing framework for drawing hierarchical graph layouts and (2) to develop a heuristic to solve the constrained crossing reduction problem. The proposed dissertation will extend the North and Woodhull (2001) research and develop a variant version of the online dynamic graph drawing system to achieve a solution that reduces the number of crossings while accommodating the set of constraining criteria proposed by North (1995).

The specific goals of this research are as follows:

1. Extend the work of North and Woodhull (2001) by developing an incremental graph drawing framework that supports the following six operations:

   a. Inserting a vertex or pseudo-vertex and a given set of edges connecting the new node to existing vertices.

   b. Deleting a node and all its incident vertices.

   c. Adding an edge.

   d. Deleting an edge.

   e. Adding a movement constraint, which restricts vertices from moving from their original positions. The value of this constraint is used to balance layout stability with readability.

    f.   Removing a movement constraint from a vertex if applicable.

2.   Formulate a parameterized equation that accommodates aesthetic criteria. This equation will factor in the number of crossings for the one-sided two-layered crossing reduction problem.

3.   Develop a relational data model to capture incremental graph layouts.

4.   Develop a heuristic to solve the crossing reduction problem.

5.   Analyze the heuristic's time and space complexity, and consider performance guarantees relative to an optimal solution.

Relevance

       Graph visualization and graph drawing play key roles in many applications across disciplines, such as relational database modeling, object oriented modeling or UML, business modeling, organizational diagrams, molecular layout, and DNA layout (Battista et al., 1999). With advances in computer hardware and the exponential growth of data, user experience with computer visualization is becoming more sophisticated. Some graph visualization applications have been ported to the Internet, whose environment is dynamic and whose users expect fast rendering of large graphs. Real-world graph editing applications like enterprise process modeling tools require more sophisticated interactions, such as inserting vertices into and deleting vertices from the graph (North, 1995) in real-time mode. More efficient and effective graph visualization algorithms are needed for visualizing larger graphs in real-time mode (Cohen et al., 1992; North & Woodhull, 2001; Buchsbaum & Westbrook, 2000; Raitner, 2004) while still fulfilling aesthetic criteria (North, 1995). Real-world graph structures are often dynamic and

updated frequently (He & Marriott, 1998), but most standard graph drawing algorithms often apply global optimization, leading to unstable graph layouts. Moreover, most of those algorithms do not support incremental updating and thus do not scale well for displaying very large data sets or for graph layouts where users need to interact with the graph in real time.

Research has been undertaken to improve graph layout stability and performance. To improve the stability of layouts, Bohringer and Newbery (1990) proposed to use constraints that are defined by the user or are based on previous layouts. Cohen et al. (1992) proposed a dynamic algorithm for drawing planar graphs for a variety of standard drawings and define a property for dynamic graph layout called *smooth update.* Luder, Ernst, and Stille (1995) presented a graph drawing application called *automatic display layout* that preserves the topology of the layout across sequential updates. He and Marriott (1998) proposed a *constrained graph layout framework* for undirected graphs and trees. North (1995) proposed an incremental graph layout system called DynaDAG, which supports hierarchical graph drawing. Brandes and Wagner (1997) formalized the aesthetic criteria notion for dynamic graph layout introduced by North (1995) based on the *random field model,* which is widely used in imaging processing. The authors then proposed a generic framework for online dynamic graph layout and experimented with the proposed framework by using spring models and orthogonal drawings. Buchsbaum and Westbrook (2000) formalized the concept of maintaining views for dynamic graph layout and proposed efficient data structures for storing the views. Their research goal was to overcome the physical limitation of computer screen size by allowing users to

focus on certain parts of the graph using expansion and contraction mechanisms while the underlying graph is subjected to edge insertions and deletions. Diehl et al. (2000) introduced an off-line dynamic graph layout algorithm called *foresighted layout* that preserves layout stability based a global graph layout that is a union of all the layouts. This approach looks ahead and renders the entire sequence of n drawings with respect to a global graph layout from a given sequence of n graphs. Raitner (2004) extended the work of Buchsbaum and Westbrook (2000) and developed similar algorithms for maintaining large hierarchical graph layouts that are also subjected to leaf insertion and deletion operations. Lee et al. (2006) presented a modified *simulated annealing* algorithm that preserves the user's mental map by adding layout stability as a factor in the cost function in the simulated annealing computation. Similarly, Frishman and Tal (2007) proposed a new algorithm based on force-directed layout for drawing online dynamic graph layouts. The authors applied degree of movement flexibility on vertices to ensure the algorithm takes into account layout stability while recalculating the next layout.

However, most techniques are applied to force-directed graph layouts and few work well with hierarchical graph layout models. Within the few proposed frameworks for drawing hierarchical graph layouts, none of the current researchers present a way to minimize the number of crossings while accounting for a set of constraints imposed by criteria in dynamic graph drawing. The original contribution of the proposed dissertation is to develop a constrained graph framework for drawing hierarchical graph layouts that includes a heuristic that reduces the number of crossings for one-sided, two-layered directed graphs while satisfying the aesthetic criteria defined by North (1995).

Barriers and Issues

In addition to the NP-completeness of the one-sided crossing reduction problem, combining dynamic criteria with the crossing reduction problem poses challenges for the dissertation work. There is an inherent trade-off between satisfying the layout stability aesthetic criterion and the crossing reduction or readability criterion. Preserving layout stability could increase the number of crossings, but reducing crossings can compromise the aesthetic criteria (North & Woodhull, 2001; Forster, 2004; Görg, 2005).

Another trade-off is between the aesthetic criteria and the space-time complexity. Satisfying the aesthetic criteria can increase the complexity of space or time or both (Cohen et al., 1992; Gansner, North, & Vo, 1993). Thus, in exploring the trade-off between minimizing the number of crossings and satisfying the aesthetic criteria, this dissertation will seek a solution that balances layout stability and drawing readability.

None of the research in dynamic graph drawing applications addresses the scalability of the use of internal data structures that capture the previous states of the graph layout. This leads to another interesting problem: how to construct an efficient graph model that enables the algorithm performance to be efficient and scalable for large graphs.

Other issues relate to the inconsistency between theoretical results and real-world applications. Results from several experiments show that some proposed heuristics solving the problem have good performance when tested using synthetic data but have poor performance using real-world data (Marti & Laguna, 2003). For instance, the median approach has better worst-case scenario efficiency than the barycenter approach

using synthetic data, but the barycenter method outperforms the median method in practice.

One issue relates to the generation of graph models used in testing. Stallman, Brglez, and Ghost (2001) mentioned that finding a collection of graph data using a random graph generator that covers different types of graph is challenging. Results from several experiments also indicate that experimental results using synthetic graph data do not necessarily reflect those of real-world graph applications (Marti & Laguna, 2003). Thus, generating graph data closely similar to real-world graph applications poses an interesting but challenging problem for measuring the performance of the proposed heuristic.

The current literature lacks detailed information for solving the layer-by-layer crossing reduction problem. Most of the literature only vaguely describes the solution for the layer-by-layer sweeping approach. The recent experiment done by Patarasuk (2004) shows that the numbers of crossings sometimes increases after a sweep but then decreases again after another sweep. Thus, the lack of clear halt criteria in the layer-by-layer crossing reduction algorithm poses an interesting problem.

Most researchers assume that minimizing the number of edge crossings will improve the readability of the layout, but the recent study by Huang and Eades (2005) shows that in some cases graph layouts with more edges crossings due to some constraints are easier to understand than the same layout with fewer edge crossings. The result of their experiments shows that human perception can be very complex. In real-world graph application, minimizing edge crossings may not necessarily improve the

users' understanding of a layout. Hence, an alternative approach that balances between minimizing the crossing number and imposing the user's defined constraints could provide a more understandable graph layout.

Formalizing aesthetic criteria based on mathematical relationships alone is not feasible, because some of the criteria are simply based on human perceptions. This unfeasibility was summarized by Knuth (1996) in his guest lecture at the Graph Drawing Conference that year. His summary was that although merging aesthetic criteria and mathematical algorithms in graph drawing creates a perception of harmony, formalizing aesthetic criteria as a mathematical equation is not feasible. The goal of this dissertation is to formalize the aesthetic criteria as a parameterized equation whose parameters are a combination of mathematical relations and human feedback.

Limitations of the Study

There are two approaches to measuring the stability of incremental graph layouts. The off-line dynamic graph drawing framework computes the next layouts based on the union of all the layouts, while the online framework computes the next layout based on the previous layout. The scope of this dissertation will be to focus on an online dynamic graph drawing framework for hierarchical directed graph layouts in which the layout stability constraint is based on the previous layout.

Most commercial data are proprietary, so the data that will be used for testing the proposed system and the heuristic for the constrained one-sided two-layered crossing reduction problem will either come from public domains or be synthetically generated.

As discussed in the Barriers and Issues section (page 9), generated graph data may not reflect closely those of real-world applications.

Real-world applications such as enterprise process modelers should support concurrent actions such as updating the graph structure. The constrained graph drawing framework proposed in this dissertation will not take into account possible concurrent execution of graph structure edits. This feature will be discussed in the Dissertation Report as a further enhancement.

Though real-world graph data have different types of shapes and sizes of vertices, for this dissertation the proposed incremental hierarchical graph drawing framework will assume that the sizes are zero and shapes are simple circles. However, the proposed framework will be designed such that the framework is extensible and accepts different sizes and shapes of vertices. The enhancement of the proposed framework will be presented in the Recommendations section of the dissertation.

Definitions of Terms

Acyclic graph: An *acyclic graph* is a simple graph that has no cycles.

Adjacency matrix: Let *G (V, E)* be a graph, and $|V|$ x $|V|$ matrix *M*. An adjacency matrix *M* is defined as follows:

$$\begin{cases} a_{ij} = 1 & \text{if there exists an edge from } v_i \text{ to } v_j \\ a_{ij} = 0 & \text{otherwise} \end{cases}$$

Adjacent vertices: Two vertices *a* and *b* are *adjacent* if they are connected by an edge *E (a, b)*.

Approximation algorithm: A design and analysis approach for solving combinatorial

optimization problems such as NP-complete or NP-hard problems. The goal of

approximation algorithms is to run in polynomial time and to provide an output

solution that is guaranteed to be close to the optimal solution.

Barycenter value of a vertex in a two-layered graph drawing: Let $G = (L_1, L_2, E)$ be a

two-layered hierarchical graph, and $D(G)$ be a drawing of G, where $u \in L_1$, N_u is

the set of vertices on layer L_2 that is adjacent with a vertex u on layer L_1. A

barycenter value of vertex u on layer L_1 is defined as the *average value* of all of

its adjacent vertices' positions. Formally, the barycenter value of a vertex u on

layer L_1 can be defined as follows:

$$\boxed{avg(u) = \frac{1}{\deg(u)} \sum_{u' \in N_u} pos(u')} \text{ (Battista et al., 1999)}$$

where *deg(u)* is the degree of vertex *u*.

Crossing number of a drawing: The *crossing number* of a drawing is the number of edge

crossings in a drawing, excluding vertex intersections. The crossing number of a

drawing is denoted as *crossing (D(G))*.

Crossing number of a drawing notation: To simplify computing the number of edge

crossings of a drawing we will define an edge crossing as an integer. If $D(G)$ is a

drawing of G and e and e' are distinct edges of $D(G)$, crossing $(e, e') = 1$ if e

crosses e', otherwise crossing $(e, e') = 0$. The edge crossing of a drawing can be

denoted as follows:

$$crossing\ (D(G)) = \frac{1}{2} \sum_{e,e' \in E} crossing(e, e')$$

Curve: A curve δ is a continuous mapping to topological space S such that $\delta : I \to S$, where I is an interval of R and S is the Euclidean plane R^2.

Cycle: A path in a graph that starts and ends at the same vertex.

Cycled graph: A graph that has one or more cycles.

Degree of a vertex: Let G be a simple graph, $v \in V$ and $e \in E$. The degree of a vertex v in the graph G, denoted as *deg(v),* is the number of edges incident to that vertex.

Directed acyclic graph (DAG): A directed graph that has no cycles.

Directed graph: A *directed graph* is a simple graph where an edge is assigned to an ordered pair of vertices. The first vertex of the ordered pair is called the *tail* of the edge, and the other is called the *head*. The direction of an edge in a directed graph drawing is represented by an arrow. A directed graph is denoted as *G (V, A)* where V is a set of vertices and A is a set of directed edges.

Directed graph with cycles: Let *G (V, A)* be a directed graph. G has a cycle if $\forall R \subset A \,|\, R$ forms a one-way loop of edges.

Distance metrics: A measurement of the distance between the location of a vertex and its previous location.

Dummy vertex: A vertex created in a process of removing edges that span more than one layer in a hierarchical graph, which makes the graph a proper hierarchical graph.

Edge spans more than a layer on layered hierarchical graph: Let *G (V, E)* be a k-layered hierarchical graph, and span $(e) = (j - i)$ be the number of layers an edge spans, where $e = (u, u')$, $u \in L_i$ and $u' \in L_j$.

Feedback arc set: Let *G= (V, A)* be a simple directed graph. The *feedback arc set* (FAS) of *G*, denoted as *R (G)*, is a set of edges (possibly empty) whose reversal makes *G* acyclic. A *minimum feedback arc set* of G, denoted as R × (*G*), is an FAS of minimum cardinality of r* (*G*) (Eades, Lin, & Smith, 1993).

Graph: A *graph G* consists of a set *V* of vertices and a set *E* of edges, where $E \subset V \times V$. Each edge has a pair of vertices referred to as its endpoints (West, 2001).

Graph density: Let *G (V, E)* be an undirected simple graph. *Graph density* is defined as a ratio of the number of edges in the graph and the maximal number of edges in the graph. Formally: $D = \dfrac{2|E|}{|V|(|V|-1)}$, where *|V|* is the number of vertices and *|E|* is the number of edges in the graph.

Incident edges: An edge *E* is *incident* to its endpoints or vertices.

Incremental graph layout: Please see definition of Online dynamic graph layout.

Independent set: Two sets *A* and *B* are said to be *independent* if their intersection *A ∩ B* = Ø, where *Ø* is the empty set. Independent sets are also called *disjoint* or *mutually exclusive*. Independent sets or disjoint sets are used in defining partite graphs (Weisstein, 2003).

Jordan arc: A *Jordan arc* is a subinterval *(c, d)* of a Jordan curve, where $a \le b \le c \le d$.

Jordan curve: A *curve* is closed or a loop if *I = (a, b)*, *a ≠ b* and δ *(a) = δ(b)*, where *δ* is defined as a curve (see definition of Curve). A *Jordan curve* is defined as a non-self-intersecting loop in a plane, which divides the plane into two disjoint regions, the inside and the outside.

K-*layered hierarchical graph:* A k-*layered hierarchical graph* is a *k*-partite graph *G (V,*

E) in which *V* is partitioned into *k*-partite sets l_1, l_2, l_3, ... l_k such that *(u, v)* $\in V$,

where $u \in l_i$, $v \in l_j$, *and i < j*. A *k*-layered hierarchical graph is drawn such that

the vertices in a given layer are drawn on a horizontal axis and the edges are

drawn as straight lines. The height of a *k*-layered graph layout is the number of

layers, which is *k*. The width of the layout is the number of vertices in the layer

that has the most vertices. Figure 2 shows a drawing of a four-layered hierarchical

graph that has a height of *4* and a width of *5*.
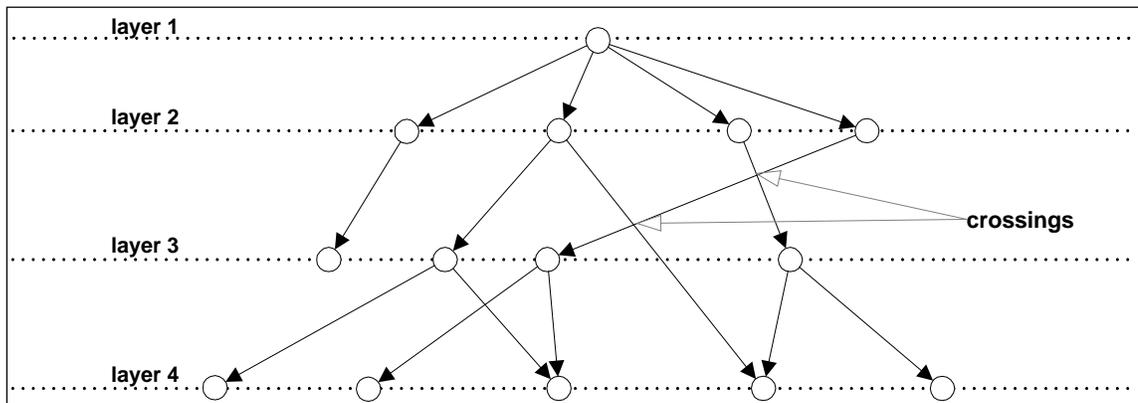
Figure 2. A 4-layered graph layout (Battista et al., 1999)

K-*partite graph:* A k-*partite graph* is a simple graph *G* whose vertices are a union of *k*

independent (possibly empty) sets of vertices such that no two vertices in the

same set are adjacent (West, 2001). Figure 3 shows a *two-partite*, or *bipartite,*

graph with two independent sets of vertices: *(a, b, c)* and *(d, e, f, g, h)*.
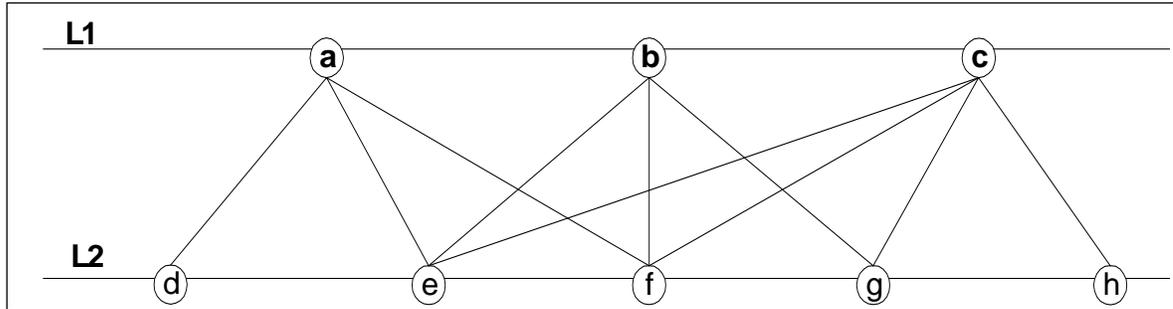
Figure 3. A bipartite graph

Lexicographical order: Given *A(a, b)* and *B(a′, b′)* are two partially ordered sets,

the *lexicographical order* of the Cartesian product of $A \times B$ is defined as follows:

(a,b) ≤ *(a′,b′)* iff *a < a′ or a = a′ and b < b′*

Median value of a vertex: Let $G = (L_1, L_2, E)$ be a two-layered hierarchical graph where

$u \in L_1$ and $u′ \in L_2$, and pos_1, pos_2 is the ordering of layers L_1 and L_2 respectively.

The median value of a vertex u on L_1 is described as follows:

If adjacent vertices of the vertex u are vertices $u′_1, u′_2, ...,u′_n$ on layer L_2, with *pos*

$(u′_1) < pos (u′_2) < .... < pos (u′_n)$, where *pos* is the ordering of vertices on a layer

and n is the number of vertices on layer L_2, the median value of vertex u, denoted

as *med(u),* is chosen as a median of all the positions of vertices $u′$ that are adjacent

to vertex u (Battista et al., 1999).

Formally: med (u) = pos ($u′_{floor(n/2)}$)

If vertex u has no adjacent vertices then *med(u) = 0*.

Mental map: A person's perception or internal representation of an area that helps

organize and interpret its information. Mental maps can be affected positively or

negatively by the stability and readability aesthetic criteria in dynamic graph

layout algorithms.

Neighborhood of a vertex v: A set of vertices adjacent to *v,* written as *N(v)* (West, 2001).

Off-line dynamic graph layout: Given a sequence of n graphs $g_1, g_2, ..., g_n$. Compute

layouts $l_1, l_2, ..., l_n$ for these graphs such that

(1) $\overline{\Delta} = \sum_{1 \leq i \leq n} \Delta(l_i, l_{i+1})$

(2) $\overline{\Gamma} = \sum_{1 \leq i \leq n} \Gamma(l_i)$

where $\overline{\Delta}$ is a deviation of all the layouts and $\overline{\Gamma}$ is defined as layout quality based

on aesthetic criteria (Diehl & Görg, 2002).

Online dynamic graph layout: Given a sequence of n graphs $g_1, g_2, ..., g_n$. Compute

layouts $l_1, l_2, ..., l_n$ for these graphs such that layout l_i is similar to l_{i+1}

Ordering of vertices on a layer in a **k-*layered hierarchical graph:*** Let *D(G)* be a

drawing of hierarchical graph *G,* $V_i = \{v_1, ..., v_{ni}\}$ are vertices of layer *i*. *pos*:V_i →

(1, . . ., n_i) is defined as a bijective function that maps vertices on layer *i* to the

drawing *D* such that *pos(v_i) < pos(v_j)* if *x(v_i) < x(v_j),* where *pos* is defined as an

ordering of vertices on layer *i* in a given drawing *D(G)* and *pos(v)* is the position

of vertex *v* on layer *i*.

Given that *u, v* are vertices on layer *i* in a given drawing *D(G),* a binary relation <

is defined as relative positions between vertices *u* and *v* such that *u < v iff pos(u)*

< pos(v).

Path: A list of vertices of a graph where each vertex except the last has an edge

connecting it to the next vertex.

Proper layered hierarchical graph: A layered hierarchical graph is *proper* if it has no

edges that span more than one layer. The top layout in Figure 4 shows a layered

hierarchical graph that is not proper because two of its edges span more than one

layer. To make a layered hierarchical graph proper, each edge in the graph that

spans more than one layer is split into multiple edges by inserting dummy vertices

into the layers. The bottom layout shows the layered hierarchical graph made

proper by splitting the two edges that span more than two layers into multiple

edges. Two new dummy vertices have been created on layer 3.

A layered hierarchical graph with edges spanning more than one layer

The layered hierarchical graph made proper by inserting dummy vertices
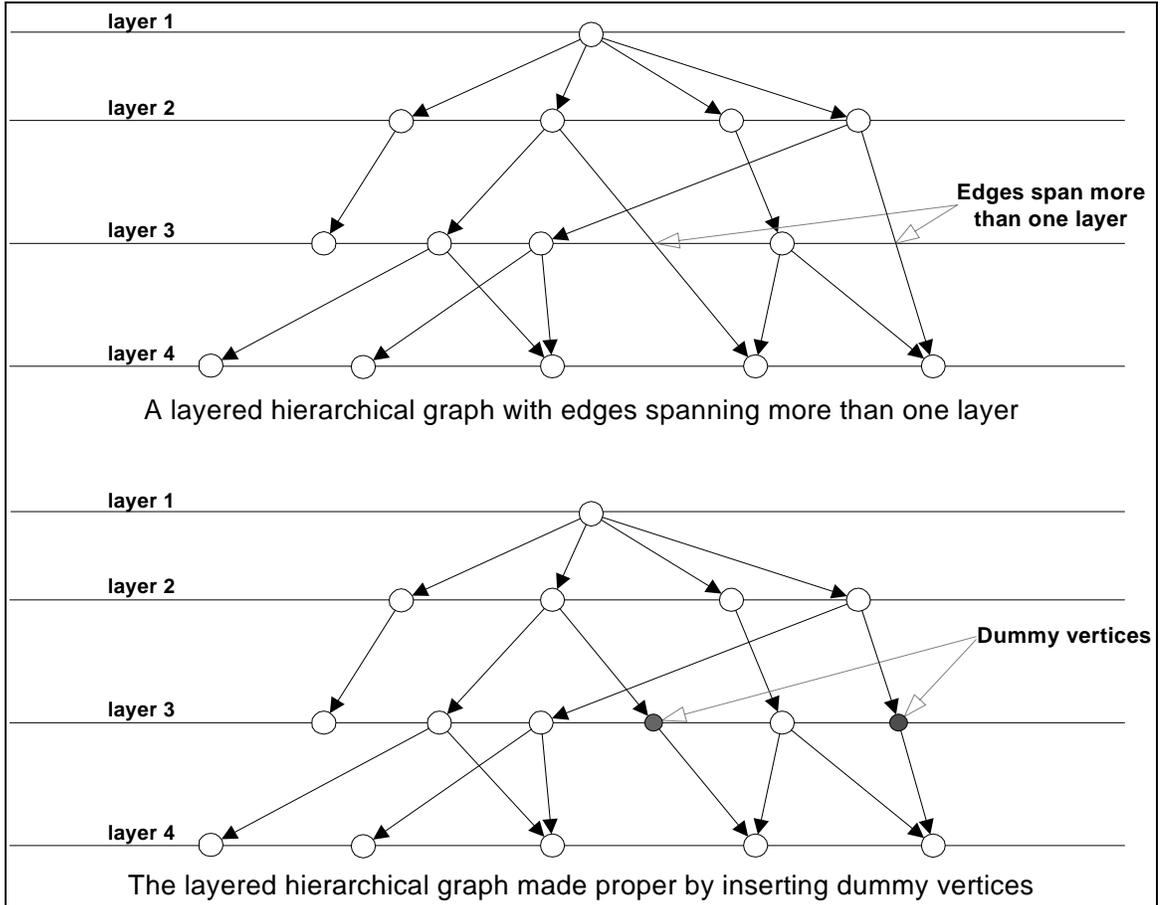
Figure 4. A layered hierarchical graph made proper by inserting dummy vertices

Quality of a layout: Let *G* be a constrained graph layout, and $l \in G(l)$ be a layout of *G*.

Then the function $Q : D(l) \rightarrow R^+$ is defined as a metric for the quality of the

layout. For instance, $Q(l) = 0$ means that *l* has minimal quality (Görg, 2005). In

terms of layout aesthetics, the metric for the quality of a layout is the number of

crossings.

Quality of incremental graph layout: This is an optimization problem with two objective

functions as follows:

(1) $\overline{\Delta} = \sum_{1 \leq i < n} \Delta(l_{i-1}, l_i)$ is minimal

(2) $\overline{Q} = \sum_{1 \le i < n} Q(l_i)$ is maximal

where $\overline{\Delta}$ is a deviation of all the layouts and $\overline{Q}$ is defined as layout quality based

on aesthetic criteria (Diehl & Görg, 2002). In terms of graph layouts, aesthetic

property (1) is equivalent to preserving the mental map of the layout and (2) is

equivalent to reducing the number of edge crossing in the layout. These two goals

often contradict each other.

R-*approximation algorithm:* A polynomial-time algorithm that produces a solution at

most *r* times the optimum for a minimization problem (Rabani, 2003).

Ratio bound performance of one-sided crossing reduction heuristics: Let G = *(L₁, L₂,*

E) be a two-layered hierarchical graph, *u, v* ∈ L_1, *pos₁, pos₂* be the ordering of

layers L_1 and L_2 respectively, and *pos₂* be held fixed. If *h* is a heuristic for solving

the one-sided crossing reduction problem, a ratio bound of heuristic *h* can be

defined as follows:

Ratio bound of $h = \dfrac{\text{opt}_h\,(G,\, \text{pos2})}{\text{LB}\,(G,\, \text{pos2})}$ (1)

In which $\text{LB}\,(G,\, \text{pos2}) = \sum_{u,v \in L_1} \min(c_{uv}, c_{vu})$ (2) as defined in the *Trivial lower*

bound of the one-sided two-layered graph crossing reduction problem definition

(1) & (2) => Ratio bound of $h = \dfrac{\text{opt}_h\,(G,\, \text{pos2})}{\displaystyle\sum_{u,v \in L_1} \min(c_{uv}, c_{vu})}$

where $opt_h(G, pos_2)$ is the minimum number of crossings produced by the

heuristic h, and $\sum_{u,v} \min(c_{uv}, c_{vu})$ is the trivial lower bound of the one-sided two-

layered crossing reduction problem.

Simple graph: A *simple graph* is a graph that has no loop or multiple edges.

Sink: a vertex that has incoming edges but no outgoing edges.

Source: a vertex that has outgoing edges but no incoming edges.

Topological sorting: Let G be a directed acyclic graph (DAG). Topological sorting is a

topological numbering of G such that every vertex is assigned a unique integer

between 1 and n (Battista et al., 1999).

The crossing number of a graph: Let G be a graph and $D(G)$ a drawing of G. The

crossing number of a graph is the minimum number of edge crossings in any of its

drawings in a plane R^2:

crossing (G) = min {crossing (D(G) | D(G) is a drawing of G}

Two-layered hierarchical graph: A *two-layered hierarchical graph* is denoted as a

triple: $G = (L_1, L_2, E)$, $(u, u') \in E$ where $u \in L_1$ and $u' \in L_2$. Figure 5 shows a

two-layered hierarchical graph.
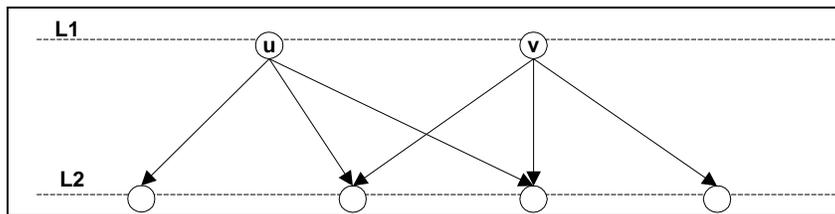
Figure 5. A two-layered hierarchical graph

The crossing number in a drawing of a two-layered graph: Let $G = (L_1, L_2, E)$ be a two-

layered hierarchical graph where pos_1 and pos_2 are orderings of layers 1 and 2

respectively. *Cross (G, pos1, pos2)* is then defined as the crossing number in a drawing of *G*.

The crossing reduction problem of one-sided two-layered graphs: Let *G* be a bipartite graph where L_1, L_2 are layers of *G* and pos_1 and pos_2 are orderings of layers L_1 and L_2 respectively. L_2 is held fixed, and let *opt(G, pos₂)* be the minimum number of crossings of drawing *D* of *G* with respect to pos_2. The crossing reduction problem is to find the minimum number of edge crossings of layer L_1. Formally:

Let $G = (L_1, L_2, E)$ be a two-layered hierarchical graph with an ordering pos_2. Find an ordering pos₁ such that *crossing (G, pos₁, pos₂) = opt(G, pos₂)*.

Hence, the minimum number of crossings of a drawing *D* of *G* is:

$$opt(G, pos_2) = min \; \{cross(G, pos_1, pos_2)| \; pos_1 \in S_{|V1|} \} \; (1)$$

where S_n is the symmetric group of all permutations on layer 1.

The crossing number of two vertices in a one-sided two-layered graph: Let $G = (L_1, L_2, E)$ be a two-layered hierarchical graph; $u, v \in L_1 \mid pos(u) < pos(v),$ C_{uv} is defined as the crossing number of edges incident with *u* and edges incident with *v*.

$$\boxed{C_{uv} = \sum_{e \in inc(u), e' \in inc(v)} crossing(e, e')}$$

Where *inc(u)* is a set of edges incident to vertex *u*.

It is known that the number of crossings between edges incident with vertex *u* and edges incident with *v* depends only on relative positions of *u* and *v* and not on other vertices (Battista et al., 1999). As illustrated in Figure 6, in the first layout *u* is placed before *v*, so C_{uv} is 1. The second layout shows that the order of *uv* is the

same even if the position of vertex *v* has moved, so C_{uv} is still 1. The third layout

shows the order between vertices *u* and *v* has been swapped and the new crossing

number C_{vu} is now 6.
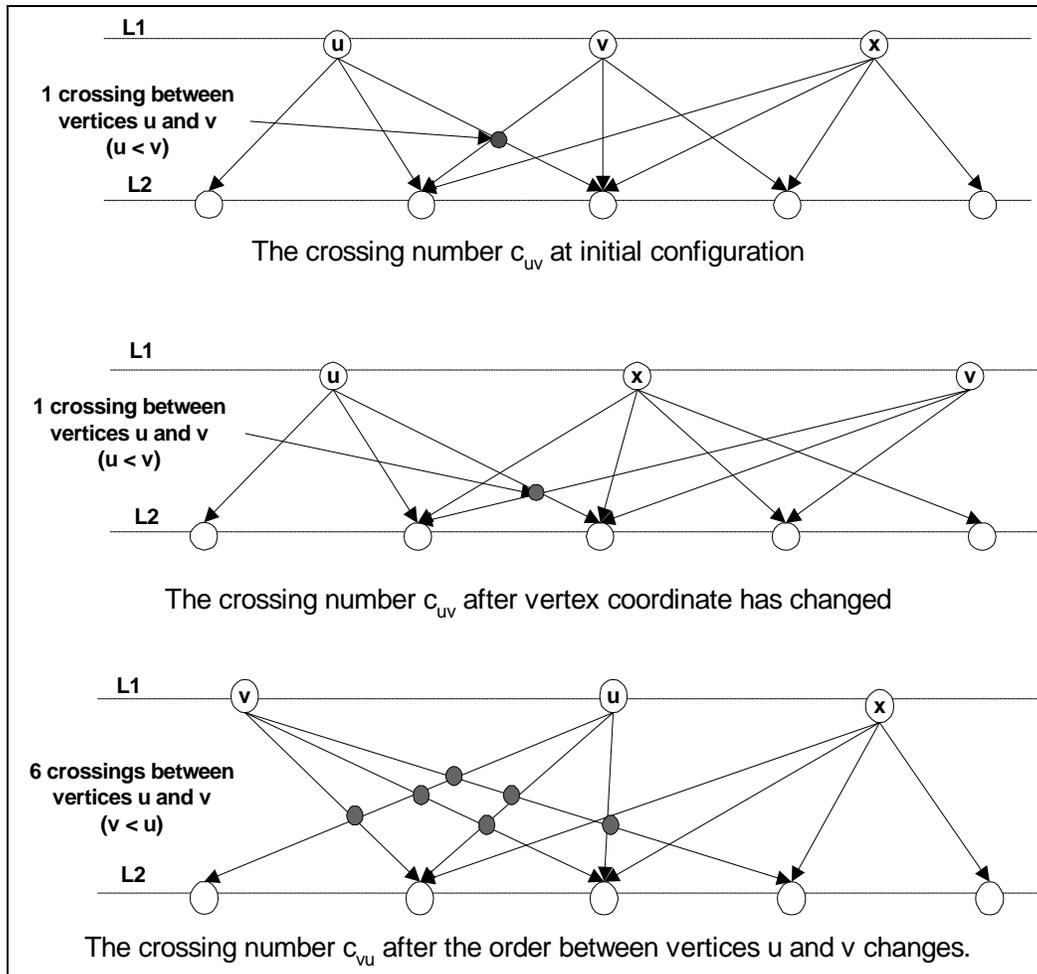
Figure 6. Crossing number of c_{uv} and c_{vu}

The *crossing number* in a drawing *D* of a one-sided two-layered graph *G* can be

defined as the sum of the number of edge crossings of all the pairs of vertices on

the layer L_1.

Formally: *crossing (D(G), pos$_1$, pos$_2$)* = $\displaystyle\sum_{u,v \in L_1 \mid pos(u) < pos(v)} c_{uv}$ (1)

Where *opt(G, pos₂) = min {cross(G, pos₁, pos₂)| pos₁ ∈ S_{|V1|} }* (2), as defined in

The crossing reduction problem of one-sided two-layered graphs.

Combine (1) and (2): $opt(G, pos_2) \geq \sum_{u,v \in L_1} \min(c_{uv}, c_{vu})$

Trivial lower bound of the one-sided two-layered graph crossing reduction problem: A

trivial lower bound of the one-sided two-layered graph crossing reduction

problem can be defined as follows:

$$LB\ (G, pos_2) = \sum_{u,v \in L_1} \min(c_{uv}, c_{vu})$$

This trivial lower bound will be used to compute the efficiency of the heuristic.

Vertex degree: The degree of a vertex is the number of edges incident to the vertex. The

degree of a vertex *v* is denoted as *deg(v)* (West, 2001).

Vertex outdgree: Let *G = (V, E)* be a directed graph; the outdgree is the number of edges

incident to the vertex and heading outward from the vertex.

Summary

Data in real-world graph drawing applications often change frequently but

incrementally. Any drastic change in the graph layout could disrupt a user's "mental

map." Furthermore, real-world applications like enterprise process or e-commerce

graphing, where data increase rapidly, demand a good response time when rendering the

graph layout in a multi-user environment and in real-time mode. Most standard static

graph drawing algorithms apply global changes and redraw the entire graph layout

whenever the data change. The new layout may be very different from the previous

layout and the time taken to redraw the entire graph increases quickly as the amount of

graph data grows. Dynamic behavior and the quantity of data of real-world applications pose challenges for existing graph drawing algorithms in terms of incremental stability and scalability.

Dynamic graph drawing algorithms have been proposed to accommodate the dynamic behaviors of real-world graph drawing applications, but those algorithms also impose several dynamic aesthetic criteria on graph layouts. The criteria improve the incremental stability of the graph layout but their layout constraints hamper the reduction of crossings. There has been little research on the problem of minimizing crossings while adhering to a set of dynamic aesthetic criteria for dynamic graph layouts.

The research of the proposed dissertation will develop a heuristic for solving the constrained one-sided crossing reduction problem based on the work by Forster (2004). The goal of the heuristic will be to find a balance between adhering to the aesthetic criteria and minimizing the edge crossings. A modified version of the online dynamic graph drawing framework proposed by North and Woodhull (2001) will also be developed to support the experiment.

The remainder of this proposal is organized as follows. Chapter 2 reviews literature in the graph drawing area that has direct or indirect influence on this research. Chapter 3 describes the methodology of the proposed constrained hierarchical graph drawing and visualization framework that is based on the work of North (1995) and describes the modified algorithm for the one-sided two-layered crossing reduction problem based on the work of Forster (2004).

Chapter 2

Review of the Literature

Introduction

The work of this research is influenced by two areas of graph drawing frameworks, namely (1) general algorithms for drawing hierarchical graph layouts and (2) dynamic graph drawing frameworks. Accordingly, the literature review will be divided into two sections: the first section reviews the Sugiyama heuristic and the second section reviews the graph drawing frameworks. The Sugiyama heuristic has four phases, each with its own domain of research. Hence within the first section the review of the literature is divided into four subsections. Each subsection reviews the key literature of each step in the Sugiyama heuristic. Tables summarizing the characteristics of the different algorithms follow the first three subsections and the second section.

The Sugiyama Algorithm

A well-known heuristic for drawing standard hierarchical graph layouts is proposed by Sugiyama, Tagawa, and Toda (1981). This heuristic has four phases, as follows:

1. Cycle removal

2. Layer assignment

3. Crossing reduction

4. Coordinate assignment

Cycle Removal

This first phase is applied when the input graph has cycles, and ensures that a directed graph is acyclic, which is required in the layer assignment phase. To make a cyclic directed graph acyclic, a set of edges is reversed temporarily so that all the edges flow in the same direction. The main problem is to choose the smallest set of edges possible to reverse. Figure 7 shows two possible sets of edges that can be reversed to make the directed graph shown in Figure 8 acyclic. Set $A= \{(9, 4), (11, 5)\}$ and set B contains all other edges. The optimal solution for the graph in Figure 7 is to reverse set A, which has fewer edges. Figure 8 shows that the directed graph is acyclic after reversing the directions of edges (9, 4) and (11, 5).
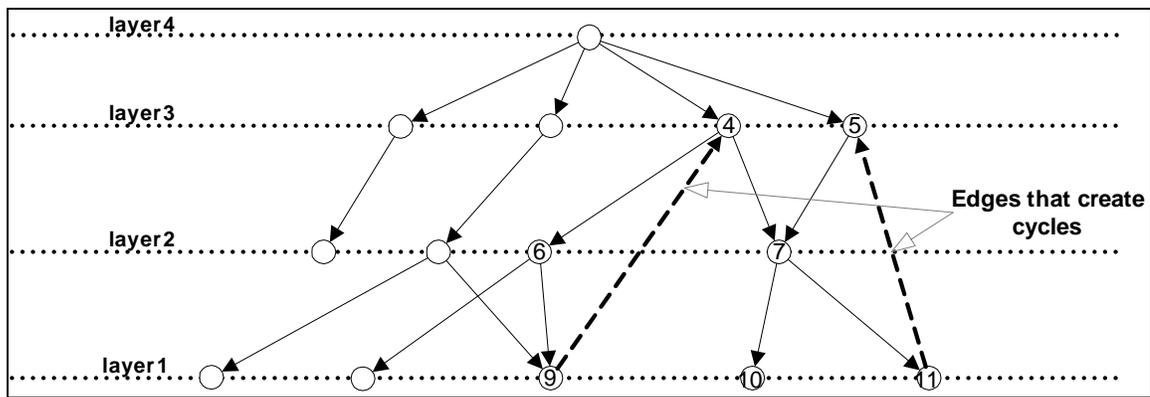
Figure 7. A directed graph with cycles
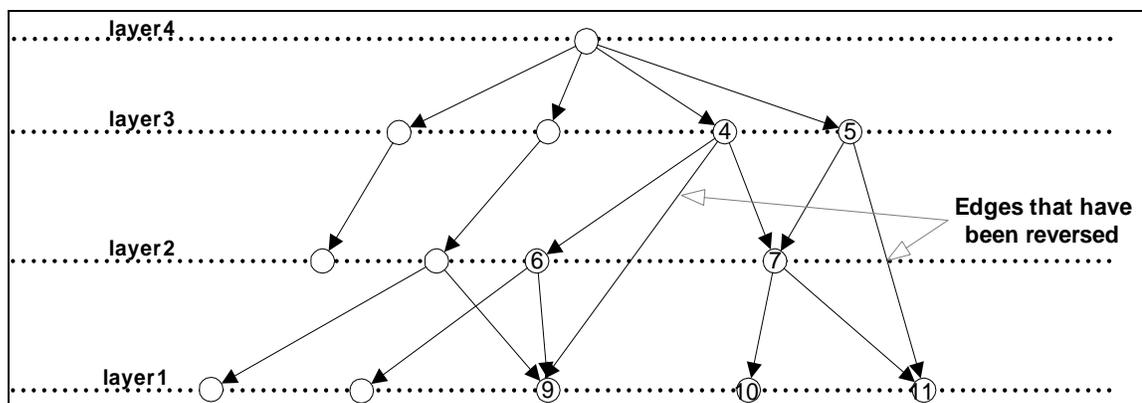
Figure 8. An acyclic directed graph after reversing the set of edges {(9, 4), (11, 5)}

A set of reversed edges in a directed graph is called a *feedback set*. This problem

relates to the well-known problem called the *feedback arc set*, which is defined as a set of

edges whose removal makes the directed graph acyclic (Battista et al., 1999). Although

the feedback set algorithm reverses a set of edges and the feedback arc set algorithm

removes or identifies a set of edges, they have the common goal of identifying the

minimum set of feedback arcs. Hence, the same algorithms and heuristics can be used for

solving both a feedback arc set and a feedback set problem. Unfortunately, finding a

minimum feedback set is NP-complete (Garey & Johnson, 1979). The common technique

for solving this type of problem is to use approximation algorithms. Three well-known

algorithmic approaches are used to find approximation solutions: *random cuts, greedy*

algorithms, and *local search*. These approaches are described in the following

paragraphs.

A simple random cuts heuristic is to choose an arbitrary ordering and then reverse

the edges that create cycles, using either *breadth-first search* (BFS) or *depth-first search*

(DFS). This heuristic is simple to implement but does not guarantee good performance

and may yield poor results (Stedile, 2001).

A well-known greedy heuristic for solving the feedback set problem is called

Greedy-Cycle-Removal (*GR*), introduced by Eades et al. (1993). Unlike previous research

on the feedback arc set problem, which could provide a good solution with a run time of

$O (|V| \times |E|),$ GR simply finds a "good" vertex sequence that has a small set of vertices

that will be reversed by going through the vertices and eliminating any that have the

maximum sum of *in* and *out* degrees. GR runs in linear time and space complexity.

Formally, the run time for the GR algorithm is $O (|V| + |E|),$ where V is a set of vertices

and E is a set of edges.

Demetrescu and Finocchi (2003) present an approximation algorithm based on the

Local-Ratio technique, which provides an approximation algorithm for the *covering*

problem. The approximation algorithm consists of two phases. The first phase searches

for simple cycles C in the directed graph. If any such cycles exist, the algorithm identifies

edges in C whose weight, denoted as ε, is minimal. Then the weight of all the edges in C

is reduced by ε and the edges with a weight of zero are removed. If no more cycles are

found the first phase terminates. The second phase is to add some deleted edges back to

the graph without re-creating cycles. The approximation ratio of the algorithm is bounded

by the length of the longest simple cycle of the directed graph. However, the proposed

algorithm worst-case run time is $O (|V| \times |E|).$

A summary of the three approximation solution algorithms is shown in Table 1.

Table 1. Summary of algorithms for solving the cycle removal step in the Sugiyama heuristic

Name	Approach	Performance	Note
BFS/DFS	Random	No guarantee	Result may be poor
Greedy-Cycle-Removal	Greedy	$O\ (|V| + |E|)$	Result is good and the run time is linear
Penalty graph	Local search	$O\ (|V| \times |E|)$	Approximation ratio ~ the longest simple cycle. The run time is not linear.

Layer Assignment

The layer assignment phase transforms a given graph structure into an acyclic directed layered graph layout by assigning vertices to layers. Due to the limitations of computer screen real estate, the goal of this step is not only to assign vertices to layers but also to ensure that the final layout has as little width and height as possible. In other words, the layer assignment problem is a two-objective function that has two optimizing variables. Unfortunately, minimizing both the width and the height of the graph layout is NP-complete (Battista et al., 1999). As a result, most of heuristics for the layer assignment problem seek to either reduce width or reduce height.

A simple algorithm called Longest Path Layering runs linearly and produces a layout with minimum height. The algorithm of the Longest Path Layering heuristic comprises two steps: (1) Place all the sinks at the bottom layer L_1, and (2) Place each remaining vertex v in layer L_{p+1}, where p is the longest path from vertex v to those vertices on layer L_1. The advantages of the Longest Path Layering algorithm are that it can be computed in linear time and it produces a drawing with a minimal number of layers. The drawback of this algorithm is that the layout could be very wide.

Assigning vertices to layers with minimum width also relates to the problem of multi-processor scheduling. The Coffman-Graham layering algorithm (Coffman & Graham, 1972) for solving multi-processor scheduling was also applied to this problem. That algorithm provides an upper bound for the width of the graph layout by accepting an input W as an upper bound value. The Coffman-Graham layering algorithm assigns vertices to layers by performing two steps: (1) Sort vertices based on their lexicographical order, which is, as defined in Chapter 1, an alphabetical order, and (2) Assign vertices to layers such that no layer has a width greater than the input W (Battista et al., 1999). Though the Coffman-Graham layering algorithm may produce layouts of a greater height than those of the Longest Path Layering algorithm, Lam and Sethi (1979) showed that in a worst-case scenario the height will not exceed twice the optimal height when $w \rightarrow \infty$, as indicated in the following equation: $h \leq (2 - 2 / w) \times h_{min}$, where w is the width of the layout and h_{min} is the optimal height.

Another aspect of the layer assignment problem is minimizing the number of dummy vertices. The number of dummy vertices created to make a directed graph proper affects the width of the layout, but most of the algorithms for layer assignment, for instance the Coffman-Graham algorithm (Coffman & Graham, 1972), fail to take into account the dummy vertices while computing the width of the layout. As a result, the actual width of the layout may be larger than expected. Unfortunately, combining the goals of minimizing the height of the drawing and minimizing the number of dummy vertices is NP-complete (Lin, 1992).

To deal with dummy vertices, Gansner et al. (1993) proposed a heuristic for solving the layer assignment problem. The proposed algorithm minimizes the number of dummy vertices by using network simplex programming to translate the layer assignment problem into an integer linear problem. The problem then is solved using a network simplex algorithm. Gansner et al. (1993) mentioned that although the time complexity of the simplex network algorithm has not proven polynomial, it can run very quickly with few iterations in practice.

Battista et al. (1999) noted that in real-world graph drawings vertices are not simple points, but are rectangles or other wide geometric shapes. Thus, the spacing between the vertices horizontally is often larger than the spacing vertically. In other words, minimizing the width is more important than minimizing the height and the Coffman-Graham algorithm is effective for drawings that are drawn from top to bottom. On the other hand, the Longest Path Layering algorithm is more effective for drawings that are drawn from left to right.

Hierarchical graph layouts tend to be drawn from top to bottom. The incremental graph drawing algorithm proposed in the dissertation will employ the Coffman-Graham layering algorithm, assigning vertices into layers.

Table 2 summarizes the characteristics of each algorithm in the layer assignment step.

Table 2. Summary of algorithms for solving the layer assignment step in the Sugiyama heuristic

Name	Approach	Performance	Note
Longest Path Layering	Produces layout with minimum height	Linear	Good for drawings that are drawn left to right. Does not take into account dummy vertices.
Coffman-Graham	Provides upper bound for layout width	Linear	Good for drawings that are drawn top-to-bottom. Does not take into account dummy vertices.
Gansner et al. (1993)	Produces minimum dummy vertices	Not linear	Though its run time is not linear, can find a solution with few iterations in real-world applications.

Crossing Reduction

This phase reduces the number of edge crossings on a proper *k*-layered hierarchical graph layout and improves its readability. As mentioned in Chapter 1, page 19, a proper *k*-layered hierarchical graph layout is a special *k*-partite graph where the vertices are assigned to horizontal layers, edges are straight and pointing in the same direction, and no edges span more than one layer.

A well-known heuristic for solving the crossing reduction problem for proper layered hierarchical graph layouts is the *layer-by-layer sweep* algorithm proposed by Sugiyama et al. (1981). This algorithm can be described as follows:

Let *G (V, E)* be a proper *k*-layered hierarchical graph with edges pointing downward. The layer-by-layer sweep algorithm considers two layers at time, starting at the top layer and sweeping downward through the layers. At each pair of layers, the ordering of the vertices on one layer is held fixed and the one-sided crossing reduction

algorithm is performed, re-ordering the vertices on the other layer to find the minimum number of crossings between the two layers. Once the algorithm reaches the bottom layer it sweeps upward layer by layer until it reaches the top layer. The algorithm continues to sweep downward then upward until the number of crossings stops decreasing.

The proper k-layered hierarchical graph layout can be reduced to a series of two-layered hierarchical graph layouts. It is observed that the number of crossings of a proper k-layered hierarchical graph layout is the sum of the number of crossings of all the two-layered layouts. Hence, the crossing reduction problem of a proper k-layered graph can be reduced to a crossing reduction problem for a two-layered graph.

There are two possible approaches to finding the minimum number of crossings for each layer in the layer-by-layer sweep algorithm. One approach, called *two-sided crossing reduction,* allows an algorithm to permute vertices on both layers (hence "two-sided") to find the minimum number of crossings. The other approach, called *one-sided crossing reduction,* holds one layer fixed and permutes the vertices on the other layer (hence "one-sided") to find the minimum number of crossings. Though in theory the first approach may produce a better result, it is best for instances of graphs that have few vertices (Junger & Mutzel, 1997). In practice, one-sided crossing reduction is employed in the layer-by-layer sweep algorithm. As mentioned in Chapter 1, the crossing reduction problem for one-sided two-layered graphs can be defined as follows: Given G (L_1, L_2, E) where an ordering pos_2 of layer L_2 is fixed, find the ordering pos_1 of layer L_1 that results in the fewest crossings.

A brute force computation for the one-sided crossing reduction problem is to compute the number of edge crossings generated by all permutations of the vertices of layer L_1 while the ordering of vertices on L_2 is held fixed. The ordering of vertices on layer L_1 that results in the fewest crossings is the optimal solution. The top layout in Figure 2 shows a two-layered hierarchical graph layout before the one-sided crossing reduction algorithm is performed. The six edge crossings are represented by the gray dots. The bottom layout of Figure 9 shows the same two-layered hierarchical graph layout after the one-sided crossing reduction algorithm is performed and the crossing number is reduced to 1.
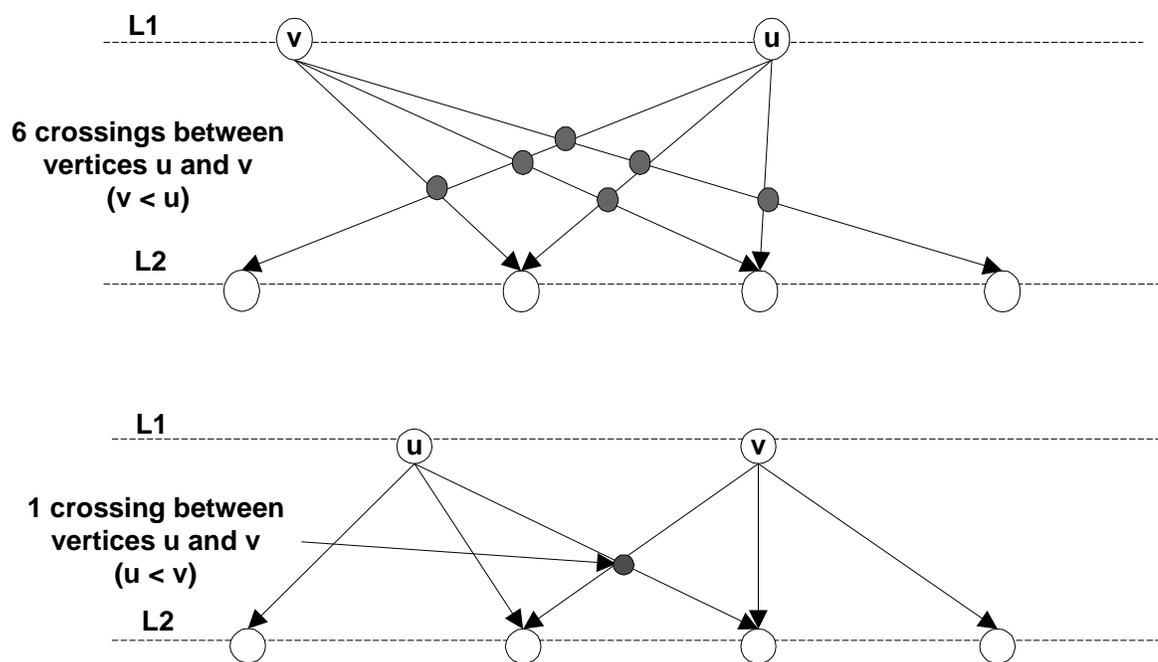
Figure 9. A two-layered hierarchical graph layout before and after crossing reduction is performed

Unfortunately, the one-sided crossing reduction problem for two-layered hierarchical graphs is NP-complete (Garey & Johnson, 1983). The brute force approach

works only with small two-layered hierarchical graphs with few vertices. Finding an optimal solution for larger hierarchical graphs requires heuristics.

The barycenter heuristic (Sugiyama et al., 1981) is well known for its simplicity and effectiveness. The algorithm reduces the number of crossings by performing two basic steps: (1) Compute a barycenter value for each vertex on layer l_i and (2) Sort vertices according to their barycenter values. The result of sorting yields the fewest edge crossings possible. Although in theory the ratio bound performance--which is a ratio of the number of edge crossings produced by the algorithm and the minimum of number of edge crossings--of the barycenter heuristic is ($\sqrt{|V|}$) (Li & Stallman, 2002), this heuristic produces a very good layout and outperforms most algorithms in practice (Junger & Mutzel, 1997).

Eades and Kelly (1986) proposed a *split* algorithm, which is very similar to a quick sort algorithm. The algorithm chooses a vertex as a pivot and then places all other vertices to the left or right of the pivot vertex depending on which way would produce fewer crossings. The step is applied recursively for all vertices on the same layer. In practice, the split algorithm is implemented in two steps: (1) Create a crossing matrix and (2) Perform the crossing reduction. The asymptotic performance of this algorithm is *O ($|V| \times |E| + |V| \log|V|$).*

Eades and Kelly (1986) also proposed a heuristic called *greedy-switch*. The algorithm scans all consecutive pairs of vertices and switches their positions if it reduces the number of crossings. The scan continues until no switching can produce fewer crossings. The asymptotic performance of this algorithm is *O ($|V| \log|V|^2$).* Junger and

Mutzel's (1997) experimental result showed that these recursive heuristics are outperformed by the barycenter heuristic and the third Eades and Kelly proposal, the *median* heuristic.

Eades and Kelly's median heuristic (1986) is similar to the barycenter heuristic. Both barycenter and median algorithms sort vertices based on their average values, but the barycenter sorts a layer's vertices according to the barycenter values, while the median heuristic sorts them according to the median values. In theory the median heuristic, with an approximation guarantee of factor of 3 of optimal, has a better ratio bound than the barycenter heuristic, whose ratio bound is $\sqrt{|V|}$ (Li et al., 2006). In practice the barycenter heuristic outperforms the median heuristic (Marti & Laguna, 2003; Junger & Mutzel, 1997).

Catarci (1988) proposed the *assignment* heuristic. The assignment problem is designed to find a best task for workers using an adjacency matrix. The author reduced the one-sided crossing reduction to an assignment problem by converting the bipartite graph data into a four-dimensional matrix. The algorithm performed well for graphs with a density greater than 30%. The run time of the assignment heuristic was defined as a ratio of the crossing number and the lower bound: $runtime = \dfrac{CN}{LB}$, where *CN* denotes the crossing number and $LB = \sum_{u,v} \min(c_{uv}, c_{vu})$, as defined in Chapter 1, is a trivial lower bound. An experiment performed by Junger and Mutzel (1997) indicated that the barycenter heuristic outperformed the assignment heuristic in many instances of graphs

with different densities, but the assignment heuristic consistently produced an attractive graph layout.

Junger and Mutzel (1997) presented an algorithm called *branch and cut*. The authors defined minimizing the number of crossings as an objective function with respect to a set of constraints. The one-sided two-layered crossing reduction problem can be expressed as linear programming (LP). The authors determined that the branch and cut heuristic can find a true optimal solution for a small graph with fewer than 60 vertices and layers with fewer than 15 vertices. For larger graphs the authors suggested using the barycenter heuristic.

Matuszewski, Schönfeld, and Molitor (1999) presented a heuristic for solving the one-sided two-layered crossing reduction problem based on a technique called *sifting* (Rudell, 1993), which reduced the number of vertices in a *reduced order binary decision diagram* (ROBDD). The sifting algorithm can be described as follows: Given a one-sided two-layered graph $G = (E, L_1, L_2)$, in which vertices on L_1 are held fixed. The sifting algorithm will choose a vertex v from L_2 and put it in a position that produces a local optimal for minimizing the number of crossings while other vertices on L_2 remain fixed. The procedure is straightforward. First, vertex v is shifted to the leftmost position by swapping with the neighbors to its left. It is then shifted to the right. Once the vertex reaches the rightmost position, vertex v is moved to a position that produces a local optimal solution, the minimum number of edge crossings. This step is done by comparing the number of crossings after each swapping. The authors showed that the sifting heuristic run-time performance is slightly better than that of the barycenter heuristic for

small, spare, two-layer graphs. However, the barycenter heuristic outperforms the sifting heuristic because the sifting heuristic run time is $O(|V|^2)$.

Based on *local search,* a common approach for improving solutions to optimization problems, Stallman et al. (2001) proposed a heuristic called *adaptive insertion,* which was a generalization of the local search approach. The basic operation of the *adaptive insertion* heuristic is to swap each vertex with its neighbors in the same layer. This operation is performed iteratively until no better result is found or fewer crossings are found. The overall asymptotic performance for each iteration was $O(|V| \times |E|)$. The experimental result indicated that the *adaptive insertion* heuristic is not scalable for large graphs.

Demetrescu and Finocchi (2003) addressed the strong relationship between the crossing reduction problem and the problem of finding minimum feedback arc sets in directed graphs. The authors showed that the number of crossings in a two-layered graph can be represented as a graph called a *penalty graph.* The authors also proved that the crossing reduction problem is equivalent to the feedback arc set problem. In the reduction, the final penalty graph, after cycle removal, represents the ordering of vertices on the layer such that the number of crossings is minimal. The authors performed several experiments with different data sets. The experimental result showed that the proposed algorithm produces fewer crossings than does the barycenter method. The drawback of this approach is that the algorithm had a time complexity of $O(|V|^4 + |E|^2)$. This approach is not scalable for large graphs.

Marti and Laguna (2003) performed extensive experiments comparing 12 well-known heuristics and two meta-heuristics. The authors concluded that for dense graphs *Tabu search* is an appropriate choice for solving the crossing reduction problem, and for sparse graphs the *GRASP* meta-heuristic produces better results than other heuristics. However, the authors also suggested that if performance is critical the hybrid barycenter or splitting heuristic is a good candidate.

Most of the research cited so far focuses on the crossing reduction problem without constraints. In real-world hierarchical graph drawing applications, users sometimes apply constraints on vertices and restrict them from changing their positions on the layers to preserve layout stability. Reducing edge crossings for one-sided two-layered graph layouts with vertex constraints is called the *crossing reduction problem for constrained one-sided two-layered graphs*. Formally, given a two-layered graph G $(L_1,$ $L_2, E)$, where L_2 is fixed, and a set of constraints $C \subseteq L_1 \times L_1$. Find a permutation of vertices on layer L_1 with few edge crossings and satisfied constraints. This problem is also NP-hard (Finocchi, 2002; Forster, 2004). A constraint $c(u, v)$ is defined such that iff pos(u) < pos(v). The constraint $c(u, v)$ is satisfied when pos(u) < pos(v), and is violated when pos(u) > pos(v).

Sander (1996) proposed a simple solution for solving crossing reduction for constrained one-sided two-layered graph layouts. The proposed algorithm first computes the barycentric of vertices. Next, it sorts the vertices based on their barycentric values with one condition: the position of a pair of vertices is swapped if and only if that pair of

vertices has no constraint or its constraint is not violated. Overall, the proposed algorithm is a barycenter heuristic with a modified sorting algorithm.

Waddle (2001) proposed a solution similar to that of Sander's (1996). After calculating the barycentric of vertices, the algorithm loops through a set of constraints. For each constraint, if the constraint is violated it will swap the barycentric value of the source with the barycentric of the target vertex. This approach ensures that sorting the vertices based on the barycentric value will not violate any constraints. However, the result showed that the produced graph layouts are worse than the graph layouts without constraints.

Finocchi (2002) proposed a heuristic by reducing the crossing reduction problem for constrained one-sided two-layered graph layouts to a *weighted feedback arc set* problem. The heuristic first constructs a *penalty graph,* which is a mapping of a one-sided two-layered graph layout into a weighted directed graph. Constraints are added as edges with infinite weight. Then the heuristic for solving the weighted feedback arc set problem is applied. The penalty graph approach produced good results with fewer edge crossings compared to the barycenter heuristic, but its performance was not as good as that of the barycenter heuristic (Forster, 2004).

Forster (2004) presented a simple algorithm that extends the barycenter heuristic. The main idea of the algorithm is as follows. Let the order of vertices be sorted from left to right based on their barycentric values. The greater barycentric value of the vertex u indicates more edges are to the right of the vertex than to its left. In the same manner, the lesser barycentric value of the vertex v indicates more edges are to the left of the vertex

than to its right. Forster (2004) proposed a way to reduce the edge crossings without violating the constraints, by placing no vertices between the two vertices that have violated constraints (pos (u) > pos (v)). This algorithm first computes the barycentric values. Next, for each violated constraint *c(u, v),* it moves all the vertices that are between the source and target vertices to the area outside. Finally, the algorithm sorts vertices based on their barycentric values. The author showed that the proposed algorithm produces a good quality graph layout and is as fast as the standard barycenter algorithm. His algorithm had a time complexity of $O(|V| \log |V| + |E| + |C|^2)$.

Table 3 summarizes the characteristics of each crossing reduction algorithm discussed in this section.

Table 3. Summary of algorithms for solving the crossing reduction step in the Sugiyama heuristic

Name	Approach	Performance	Note
Barycenter	Sorting vertices	Near linear	Outperforms most of algorithms in real-world applications
Split (Eades & Kelly, 1986)	Reorder vertices through a pivot point	$O(\lvert V \rvert \log \lvert V \rvert)$	Good performance comparing to barycenter and median
Greedy-switch (Eades & Kelly, 1986)	Scan vertices and compare the crossing numbers	$O(\lvert V \rvert \log \lvert V \rvert^2)$	Runs effectively in real-world applications
Median (Eades & Kelly, 1986)	Sorting vertices	Near linear	Outperforms barycenter in theory but is outperformed by barycenter in real-world experiments
Assignment (Catarci, 1988)	Assignment	$\rho = \dfrac{CN}{LB}$	Efficient for layouts whose edge density is greater than 30%. However, it is not as efficient as barycenter in real-world applications.
Branch and cut (Junger & Mutzel, 1997)	Linear programming	Not linear	Finds true optimal solution for a graph with fewer than 60 vertices.
Based on sifting algorithm (Matuszewski et al., 1999)	Reduced order binary decision diagram	$O(\lvert V \rvert^2)$	Outperformed by barycenter heuristic
Adaptive insertion (Stallman et al., 2001)	Local search	$O(\lvert V \rvert \times \lvert E \rvert)$	Not scalable for large graphs
Penalty graph (Demetrescu & Finocchi, 2002	Induce as a feedback arc set problem	$O(\lvert V \rvert^4 + \lvert E \rvert^2)$	Provides better drawing with fewer crossings, but is not scalable for large graphs
Modified Barycenter (Forster, 2004)	Sorting vertices	$O(\lvert V \rvert \log \lvert V \rvert + \lvert E \rvert + \lvert C \rvert^2)$	Provides layout as good as those of other complicated algorithms, but with a better run time
Modified Barycenter (Sander, 1996)	Sorting vertices	Near linear	Result are sometimes not as good as layouts without constraints
Modified Barycenter (Waddle, 2001)	Sorting vertices	Near linear	Results are worse than those of layouts without constraints

Coordinate Assignment

In this final phase in the Sugiyama heuristic, vertices are assigned horizontal coordinates. Graph edges should be short and straight (Gansner et al., 1993). A common approach for solving this problem is the *Quadratic Programming Layout Method* proposed by Sugiyama et al. (1981). The problem is defined as a quadratic objective function with respect to a set of constraints. Unfortunately, solving this problem using linear programming is computationally expensive due to the size of the matrix. Gansner et al. (1993) presented a heuristic for solving this problem. The heuristic performance is good but it is hard to program and the layout sometimes is not pleasing (Gansner et al., 1993).

Incremental Graph Drawing Systems

Although standard graph layout algorithms have been well studied in the past decade, the growth of the Internet and the increasing amount of data in enterprise applications such as process modeling tools have posed challenges for standard graph layout solutions in terms of graph stability and scalability, as indicated in Chapter 1. To keep up with real-world application concerns, dynamic graph layout heuristics have been proposed in recent years.

Bohringer and Newbery (1990) addressed two issues with standard graph layout algorithms. The authors pointed out that without user intervention or user predefined constraints, automatic graph layout algorithms cannot ensure the semantic meaning of the layout will be preserved. The other issue is that most standard graph layout algorithms do not take into account previous layouts when computing the next layout. Thus, a new

layout may look much different from previous layouts and confuse the users. Bohringer and Newbery proposed to use layout constraints to improve the stability of layouts. The constraints can be defined by the user or are based on previous layouts. The research showed that the proposed constrained graph layout does improve stability but the system needs improvements in efficiency and scalability. Also, the proposed system did not address the constrained crossing reduction problem.

Cohen et al. (1992) suggested that a good graph drawing system should support two important characteristics, namely (1) good performance when restructuring the graph layout, and (2) the ability to maintain the stability of the layout by not changing the layout drastically. They proposed a generic framework for drawing planar graphs for a variety of standard drawings, especially trees and series-parallel digraphs. The authors also defined a property for dynamic graph layout called *smooth update.* This property represented the stability of the graph layout, which is later formalized by North (1995) in his proposed dynamic graph drawing framework.

Luder et al. (1995) presented a graph drawing application called *Automatic Display Layout* (ADL), which preserves the topology of the layout across sequential updates. The authors defined a term, *topological consistency,* which is a measure of how consistent the graph layout is with preceding layouts. The authors considered the problem to be a combinatorics optimization problem and defined a cost function that includes static and dynamic constraints. Static constraints represent the aesthetic criteria and dynamic constraints represent the changes to the layout with respect to the previous layout. Their experience showed that the system can handle a graph of up to 50 vertices.

However, the ADL system did not address how to minimize the number of edge crossings.

North (1995) formalized the notion of smooth update and dynamic graph layout stability. The author defined three aesthetic criteria for measuring the effectiveness of a dynamic graph layout: consistency, stability, and readability. *Consistency* means that the layout should adhere to the predefined business rules for a domain, *stability* requires minimal changes between successive layouts, and *readability* helps make the layout easier to comprehend. Addressing aesthetic criteria for dynamic graph layout, North (1995) proposed an incremental graph drawing system called DynaDAG based on the Sugiyama heuristic. His proposed framework preserved topological and geometrical stability during dynamic operations by applying constraints to each of the four steps in the Sugiyama heuristic discussed above. The experimental results on small graph data indicated that DynaDAG produced consistent layouts. However, scalability and constrained crossing reduction were not addressed in the DynaDAG framework.

Ryall, Marks, and Shieber (1997) proposed a constraint based drawing editor called *GLIDE* (Graph Layout Interactive Diagram Editor), which allowed users to produce small or medium diagrams while the system maintained topological stability. The GLIDE system enabled users to interact with the system in real time and provided a set of hints called *Visual Organization Features,* a predefined set of common standard vertex placements. The GLIDE system used Hook's Law to compute the graph layouts with respect to a set of constraints. The GLIDE system supported constrained graph layouts but not hierarchical graph layouts.

Extending the notion of the dynamic graph layout formalism proposed by North (1995), Brandes and Wagner (1997) proposed a generic framework for online dynamic graph layout that used a *random field*. Layout models were defined in terms of the random field, which assigned probabilities that reflected the models' conformance with the layout goals. The authors then used a Bayesian decision system to solve this problem. The authors experimented with this framework on *spring model* and *orthogonal* drawings and concluded that the proposed framework can be adapted to other types of graph layout. However, the seminal work did not present the result of the experiment in term of efficiency and performance.

He and Marriott (1998) addressed the problem with current graph layout algorithms: most of the existing algorithms were not designed for interactive graph drawing applications for two reasons. The first is that existing algorithms do not adhere to the criterion that the graph layout should preserve the user's mental map by not being altered too much. The second is that existing algorithms are quite restricted in how graphs are laid out; the algorithms are not flexible and do not enable the application to apply constraints on layout. The authors also proposed four mathematical models for a constrained graph layout framework, where three models are for undirected graphs and one is for trees.

Diehl et al. (2000) pointed out the disadvantages of graph animation and online dynamic graph layout. Graph animation technique simply shows how vertices are moved to their new positions but does not necessarily preserve the metal map. Though incremental or online graph layout does preserve layout stability, each layout is based on

the previous layout, so in a worst-case scenario maintaining an incremental graph layout involves computing the layout of the whole graph. The authors introduced an off-line dynamic graph layout algorithm called *foresighted layout* that preserved the mental map of the graph layout based on a global graph layout structure. This approach looks ahead and renders the entire sequence of *n* drawings with a respect to a global graph layout from a given sequence of *n* graphs with an assumption that the entire graph is known in advance. Görg, Birke, Pohl, and Diehl (2004) extended the foresighted layout framework in orthogonal and hierarchical graph layout. The result of the experiment indicated the framework is extensible to other types of graph layouts, but the authors admitted that it is difficult to apply a foresighted layout framework with graph layout models that are constructed through multiple phases, such as hierarchical graph layouts. Foresighted layout also did not address efficiency and scalability.

To improve the efficiency and performance of the DynaDAG framework (North, 1995), North and Woodhull (2001) proposed an online hierarchical graph drawing system. The proposed system is a client/server model that allows the client to update incrementally using a messaging protocol. To preserve layout stability the server maintains a shared graph model and updates the model upon client requests and in accordance with the constraints imposed by aesthetic criteria. To apply the constraints at each step of the Sugiyama algorithm, the authors defined an objective function with a set of constraints for each step and used a *simplex network solver* to solve the problems. Unfortunately, the online hierarchical graph drawing system did not address the constrained crossing reduction problem.

Lee et al. (2006) proposed an algorithm that preserves the mental map for general graphs based upon Davison and Harel's (1996) *simulated annealing* graph drawing algorithm. The modified simulated annealing algorithm included six aesthetic criteria defined by Bridgeman and Tamassia (2002) to reflect the user's mental map. The algorithm has three phases. The first phase is to apply the original simulated annealing algorithm to draw graphs. The second phase is to modify the graph slightly. The third phase is to redraw the graph subject to aesthetic criteria. The authors mentioned that this approach is flexible because it allows the end user to adjust the relative weight of each constraint in the algorithm.

Frishman and Tal (2007) proposed an efficient and scalable new algorithm based on directed force layout for drawing online dynamic graphs. This algorithm computed the layouts using a global layout structure. The authors noted that by moving the main algorithm executions from the computer's central processing unit to its graphics processing unit the algorithm was faster than the conventional directed force algorithms. Also, the quality of the generated layouts was as good as that of those algorithms.

Table 4 summarizes the characteristics of each dynamic graph layout framework.

Table 4. Incremental graph drawing frameworks

Name	Approach	Graph Type	Note
Bohringer and Newbery (1990)	Online dynamic graph layout	Generic graphs	Addresses layout stability, but the framework is not scalable
Cohen et al. (1992)		Tree, series-parallel digraphs	

Name	Approach	Graph Type	Note
Luder et al. (1995)	Online dynamic graph layout	Generic	Provides interactive graph drawing environment. However, it can handle a graph with only up to 50 vertices.
Ryall et al. (1997)			Interactive diagram editor for drawing small graphs
Brandes and Wagner (1997)		Generic. Applied to spring and orthogonal graph layouts.	
He and Marriott (1998)	N/A	Undirected graphs and trees	Provide mathematical models
Diehl et al. (2000)	Off-line	Generic, orthogonal, hierarchical graph layouts	Preserves the mental map using global graph layout. Animates the entire sequence of layouts.
North (1995); North and Woodhull (2001)	Online dynamic graph layout	Hierarchical directed graphs	Uses a relational database. Efficient and scalable, but does not address the constrained crossing reduction problem.
Lee et al. (2006)	Online	Simulated annealing	Allows users to adjust the relative weight of aesthetic criteria
Frishman and Tal (2007)		Directed force	Very scalable and effective for directed force graph layouts

Contribution to the Field

The work of this dissertation will make several contributions to the field of graph

drawing, namely (1) extending the online dynamic graph drawing framework (North &

Woodhull, 2001) by developing a framework for drawing and visualizing hierarchical

directed graphs that supports large graph drawing and visualization efficiently, and (2) developing a method to solve the constrained crossing reduction problem for dynamic hierarchical graph layouts.

Summary

This section reviews the key literature in two related areas of graph drawing frameworks, namely (1) the Sugiyama heuristic with associated algorithms, and (2) incremental graph drawing frameworks. The review of the literature shows that most algorithms for solving the one-sided crossing reduction problem do not take user constraints into account. Also, there has been progress in incremental graph drawing frameworks for hierarchical graph layouts, but some of the user constraints such as stability criteria have been simply defined as a pure heuristic because too few experiments have measured how humans understand graph layouts. More recent research has paid attention to constraints that are defined by users. A recent study by Huang and Eades (2005) showed that user constraints do provide better readability for readers even if the resulting layouts may produce more edge crossings than layouts that do not capture user constraints. Other research like North's (1995) showed that user constraints do help to ensure that graph layouts reflect graph semantics. However, none of the studies has addressed constrained crossing reduction in dynamic graph layouts, which is important to their comprehensibility. Furthermore, most current proposals have been limited to graphs with fewer than 50 vertices. As data grow quickly and the associations become more complicated, such as in Internet network and enterprise process modeling, a solution for rendering large graphs in a real-time environment becomes necessary.

Chapter 3

Methodology

Introduction

The work of this dissertation will include four main tasks, namely (1) develop a mathematical model representing the aesthetic criteria constraints for incremental hierarchical graph layout, (2) design and develop a framework for drawing and displaying hierarchical directed graphs by extending the online graph drawing framework developed by North and Woodhull (2001), (3) develop a heuristic for the constrained crossing reduction problem for one-sided two-layered graphs based on the work of Forster (2004), and (4) evaluate the asymptotic complexity and efficiency of the new heuristic. The following four paragraphs detail these tasks.

1. The first task is to develop a formal model for incremental graph layout. The objective of this model is to balance layout stability with readability criteria. This task will be based on the work of North and Woodhull (2001) and Görg (2005).

2. The second task is to design and develop a constrained hierarchical directed graph drawing and visualization framework that will extend and enhance the online graph drawing framework proposed by North and Woodhull (2001). The newly developed framework will include new functionality. The first function is to preserve the states of the incremental graph layouts in a relational database. This enables the new framework to support version control of graph layouts, which is important in real-world interactive graph applications such as enterprise process modeling systems.

The second function will decouple the visualization component from the editing component. This separation will enable the new framework to render large graph layouts and to support concurrent users who view the layouts in real-time environments like the Internet. The third function of the new framework will enable end users to input constraints to the layouts. These user constraints will influence the design of an algorithm that reduces the number of edge crossings.

3. The third task is to develop a modified version of the Sugiyama heuristic for updating the graph model, especially the constrained crossing reduction algorithm for one-sided two-layered graph layouts. The proposed crossing reduction algorithm will incorporate user constraints to ensure graph layout stability. The goal of the proposed algorithm is to find the optimum balance between layout stability and readability. The algorithm will be based on the work of Forster (2004).

4. The fourth task is to evaluate the performance and efficiency of the new algorithm. This will involve collecting graph data from the public domain, generating graph data synthetically, analyzing the data asymptotically, and measuring the performance and efficiency of the heuristic against existing heuristics.

Chapter Layout

As the work of the proposed dissertation will be based on the work of North (1995) and Forster (2004), the chapter first reviews the aesthetic criteria for hierarchical graph layouts. Second, it reviews the research that contributes to the work of the dissertation, as shown in Figure 0. Reviewing North and Woodhull's (2001) online graph drawing framework and aesthetic criteria provides a foundation for the design of a formal

model for incremental graph layout. Third, it reviews the standard Sugiyama heuristic, which is a foundation of the main algorithm in the proposed constrained graph drawing framework. Fourth, it discusses the development of a new incremental graph drawing system. Next, it discusses a modified version of the Sugiyama heuristic for updating graph layouts. Finally, the chapter describes experimental procedures and ends with a chapter summary.
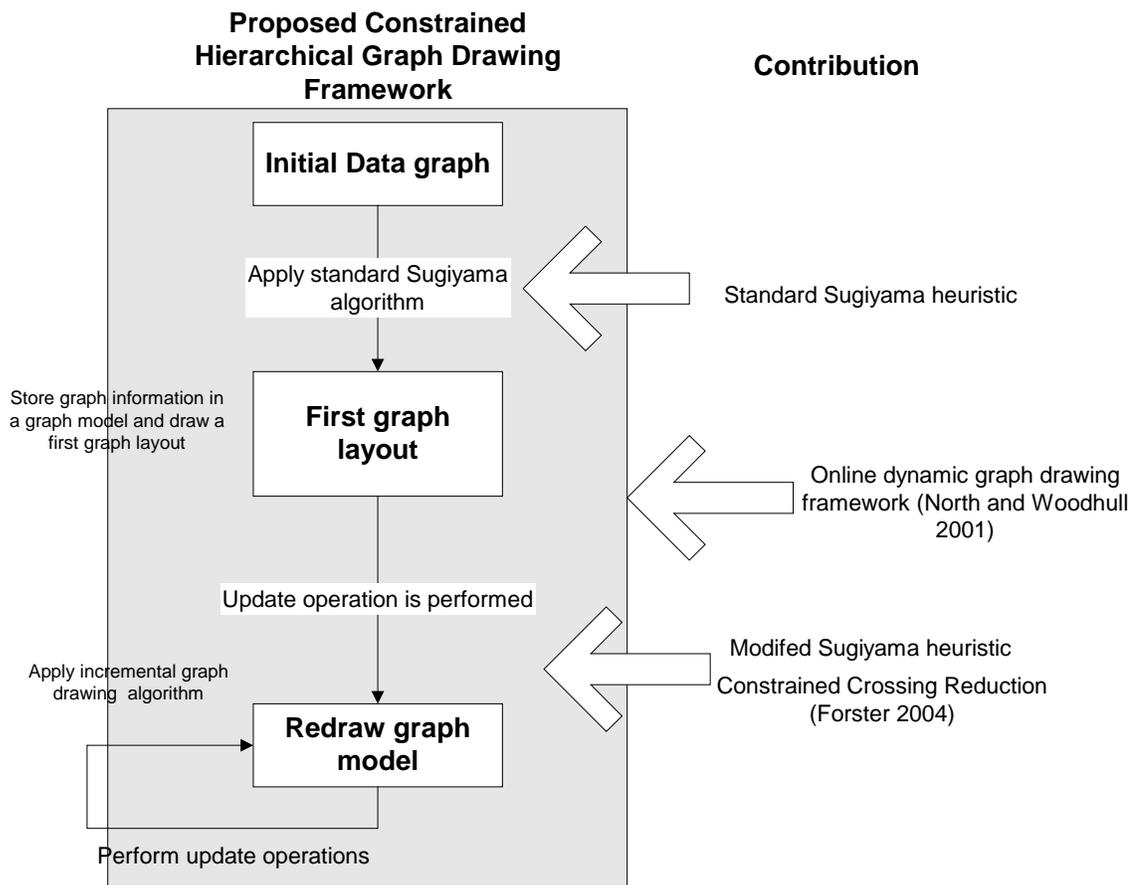
Figure 10. Proposed constrained hierarchical graph drawing system and contributed research

Assumptions and Standard Notations

For conciseness, the proposed constrained incremental graph drawing and visualization framework will be denoted as the *constrained graph drawing framework* and will be abbreviated as CGDF.

Aesthetic Criteria for Directed Hierarchical Graph Layouts

Gansner et al. (1993) listed several principles for drawing good hierarchical graph layouts. These principles are described as follows:

- *Consistency*: Edges point in the same direction. For instance, graph layouts flow from top to bottom. This aesthetic criterion is the most important for drawing directed hierarchical graph layouts because it is a fundamental characteristic of them.

- *Minimize the number of edge crossings.*

- *Keep edges short*: Short edges are easier to relate to associated vertices.

- *Keep the layout symmetrical if possible:* Edge lengths should not differ drastically.

In addition to these basic aesthetic criteria for drawing a good hierarchical graph layout, Battista et al. (1999) discussed three important requirements of a hierarchical graph layout, which are as follows:

1. The layout width and height should be as small as possible due to the constraints of screen real estate. As mentioned in Chapter 2, minimizing both width and height is NP-complete. However, as Battista et al. noted, in real-world graph drawings vertices are not simple points, but are rectangles or other geometric shapes, which tends to result in greater spacing between the vertices horizontally than vertically. In other words, minimizing the width is more important than minimizing the height.

2. The layout should be proper; i.e., no edges should span more than one layer. This requirement is to keep edges as short as possible.

3. The number of dummy vertices that are generated by making the layout proper should be as small as possible to minimize layer width.

Aesthetic Criteria for Incremental Graph Layouts

Although the aforementioned aesthetic criteria are adequate for drawing a hierarchical directed graph layout, incremental or dynamic graph layouts, whose goal is to preserve layout stability during incremental changes (North, 1995; Miriyala & Tamassia, 1993; He & Marriott, 1998; Luder et al., 1995; Cohen et al., 1992; Görg, 2005), require additional aesthetic constraints.

Unlike static graph layouts, in incremental graph layout an input graph G is considered as a series of graphs $g_1, g_2, \ldots\ldots\ldots, g_n$. The generated drawings of these successive versions of G is also a series of drawings $l_1, l_2, \ldots\ldots\ldots, l_n$ (North, 1995), where each l_i drawing is a result of update operations such as deleting or inserting vertices or edges. By making l_{i+1} resemble L_i, the incremental graph layout satisfies the following important requirements for good graph visualization (North, 1995):

1. Maintain layout stability.

2. Make changes locally.

3. Enable the layout potentially to be updated quickly.

The first two requirements ensure the graph layout preserves the mental map and helps users visualize the layout effectively, and the third ensures the system performs

efficiently. Based on these requirements, North (1995) formalizes three aesthetic criteria for drawing incremental graphs. In order of importance, these are:

1. Consistency

2. Stability

3. Readability

The consistency criterion is the same as the aesthetic criteria mentioned for static graph layouts (Gansner et al., 1993) and is the most important because it reinforces the uniqueness of the layout, such as all edges point in the same direction, all vertices are placed in a straight line, and edges should be short and not span more than one layer.

The stability criterion ensures that the user's experience with the layout is not disrupted as the graph is updated. According to North (1995) this criterion is purely heuristic because too few experiments have been done to provide conclusions about how humans read graph data and maintain mental maps effectively. The recent study by Huang and Eades (2005) showed that in some cases user constraints have a better stabilizing effect on the layout even if that layout has more edge crossings than an unconstrained layout has. Though the experiment does not cover all possible scenarios, it provides a first glance at how humans read graphs. Based on the result of the experiment, a constrained dynamic graph drawing system should give user constraints a higher precedence than the number of crossings in designing a method of solving the crossing reduction problem. North (1995) observed that the stability of the vertices is more important than that of the edges, so it is more crucial to have a higher degree of constraint of movement on vertices than on edges in designing update operations.

The readability criterion preserves drawing quality by, for example, minimizing the number of edge crossings. This criterion often conflicts with the stability criterion as discussed by Görg (2005). In his seminal dissertation *Offline Drawing of Dynamic Graphs,* Görg (2005) stated that drawing quality or local quality of the layout conflicts with global quality or layout stability. Görg (2005) pointed out that optimizing both goals at the same time is not possible because achieving high drawing quality may destroy layout stability, and improving layout stability decreases drawing quality by applying too many constraints on the layout algorithms. Thus, the goal is to find an optimal trade-off solution. The optimal solution of our proposed constrained incremental graph drawing framework will find a balance between local drawing quality and global layout stability.

The aesthetic criteria described in this section will influence the design of the constrained incremental graph drawing framework by being utilized in developing the formal layout model. The next section reviews the research on the constrained graph drawing framework. This review helps lay a foundation for our work of building a constrained graph drawing framework and solving the crossing reduction problem for one-sided two-layered graph layouts.

Related Research

The Standard Sugiyama Heuristic

This section reviews the standard Sugiyama heuristic, which will be used to build initial graph data because these data often already exist in other forms or formats. To update the graph model our proposed framework will employ a modified version of the Sugiyama heuristic, described in a later section.

In Chapter 2 the standard Sugiyama heuristic and the algorithms used in each phase of the heuristic were introduced. This section presents the pseudocode of algorithms that will be employed in the modified Sugiyama heuristic for each of the four phases in drawing constrained incremental graph layouts.

Phase 1: Cycle Removal

In Chapter 2 the brute force algorithm, the penalty graph (Finocchi, 2002), and the Greedy-Cycle-Removal algorithm (Eades et al., 1993) were introduced. Among these algorithms, Greedy-Cycle-Removal is a good choice because it produces a good result and its run time is linear. The constrained incremental framework will use this algorithm to reverse any cycles temporarily. Greedy-Cycle-Removal is a modified topological sort algorithm that sorts vertices into a sequential list based on topological ordering. The main characteristic of the algorithm is to select the vertex to be removed from G and to choose a list to add it to (Eades et al., 1993). The pseudocode of the Greedy-Cycle-Removal algorithm can be described as follows: First create two empty lists, namely S_1 and S_2. While the graph is not empty, sources are appended to S_1 and sinks are inserted into the S_2. Next, the algorithm calculates the delta between the outdegree (number of outgoing edges) and indegree (number of incoming edges) of the remaining vertices. The vertex with the largest delta value is appended to S_1. Finally, S_2 is concatenated to S_1 to create a sequence of vertices.

Table 5 shows the pseudocode of the Greedy-Cycle-Removal algorithm.

Table 5. Pseudocode of the Greedy-Cycle-Removal algorithm (Eades et al., 1993)

```
procedure (G: digraph,)
{
    Let s₁ be an empty list //Source
    Let s₂ be an empty list //Sink

     while (G not empty) {
          while (G contains a sink) {
                choose a sink u;
                s₂ <- s₂ + {u}; //Insert u to s₂
                G <- G - {u}; //Remove u from G


           }
          while (G contains a source) {
                choose a source v;
                s₁ <- s₁ + {v}; //Append v to s₁
                G <- G - {v}; //Remove v from G
          }
        //If there are vertices with both
        //incoming and outgoing edges,
        //compute the difference between outdegree
        //and indegree.
        //Find the vertex with largest delta
        //and append to s₁
        if (G not empty) {
            delta(u) <- outdeg(u) - indeg(u);
            choose a vertex u :delta(u) is maximum;
            s₁ <- s₁ + {u};  //Append u to s₁
            G <- G - {u}; //Remove u and all its incident
                          //edges from G
        }
    }  //End while loop
    //Concatenate S₂ to S₁ to form S, a vertex sequence
    S <- S₁ ∪ S₂;
}
```

Phase 2: Layer Assignment

As discussed in Chapter 2, although the Longest Path Layering algorithm is simple and has a linear run time, that algorithm could produce a very wide layer. On the other hand, the Coffman-Graham layering algorithm (Coffman & Graham, 1972) also runs in linear but limits layout width. According to Battista et al. (1999), vertices in real applications can have different shapes and sizes so minimizing the width is more

important than minimizing the height. Hence, the proposed graph drawing framework

will use the Coffman-Graham algorithm for assigning vertices into layers. This algorithm

comprises three steps. First, it assigns positive integer labels to vertices based on

lexicographical order; second, it sorts vertices into a linear list based on their integer

labels; third, it assigns the vertices to layers and ensures that the width of each layer is not

larger than the predefined value W.

The first step of the Coffman-Graham algorithm is described as follows: Initially,

all vertices are unlabelled. First, the algorithm randomly selects a source and assigns an

integer label 1 to that vertex. Then it loops through the remaining sources and assigns

integers labelled $2,3, \dots k$ to each source. For the remaining vertices in G, it performs the

following procedure for assigning integer labels to vertices: First, it selects a set R of

unlabelled vertices that have no unlabelled predecessors. Second, it sorts the set R based

on the lexicographical order of the set of predecessors' labels. Next, the procedure loops

through the set R, increments value k by 1, and assigns integer label k to vertices. This

procedure is performed until all vertices have labels.

The second and third steps are to sort vertices and assign them to layers as

follows: First, the algorithm sorts vertices based on their integer labels, and then it

assigns vertices into layers, ensuring no layer receives more than a predefined W value.

The procedure assigns vertices starting from the bottom layer l_1 and proceeds to the top

layer l_n as follows: First, it assigns all sinks to layer l_i ($1 \leq i < k$). If the width of layer L_i

is larger than W (a predefined value), then the procedure increments i by 1 and continues

this step until all the sinks are assigned to layers. To fill a layer l_k ($i < k < n$), the

algorithm selects a vertex u that has not been placed in a layer yet and all of *its* successors ($S(u)$) have been placed in one of the layers $l_1, l_2, ...l_{k-1}$. If there is more than one such vertex, the procedure will select the vertex with the largest label. If there is no such vertex or the width of layer l_k is larger than W, it proceeds to the next layer l_{k+1}. This step is performed until all vertices are assigned to appropriate layers.

Table 6 shows the pseudocode of the Coffman-Graham algorithm.

Table 6. Pseudocode of the Coffman-Graham algorithm (Battista et al., 1999)

```
procedure (G: reduced graph, W:positive integer)
{
     Let Π : v → {0,n} be integer label of v
     Let N(v) be a lexicographical order of v
     Let P(v) be predecesors of v
     Let S(u) be successors of u
     Let R(v) be a set of v
     Let i be integer: i ← 0
     Let U <= {} //be an empty set

     //Step 1: assign integer labels to vertices. All vertices are unlabelled.
         for (each u in G) {
             if (P(u) == 0){ //Select all the sources and isolated vertices
             i <- i + 1; //Increment i by 1
             Π(u) <- i;  //Assign an integer label i to u.
             U <- U + {u}; //Add v to U.
             G <- G - {u}; //Remove v from G.
           }
      }

     while (G not empty) {
       //Choose set R(v) : unlabelled v that has no unlabelled predecessors.
        foreach ( v in G) {
          if (Π(v) == 0 && P(v) are labelled) {
             R <- R + {v}; //Add v to R.
             G <- G - {v}; //Remove v from G.
          }
        } //End foreach loop
        //Sort the set R based on lexicographical order of the P(v) labels.
        //Assign label i to vertices and add them to the set U.
        foreach ( v in R) {
          i <- i + 1;
          Π(u) <- i;
          U <- U + {v}; //Add v to U.
          R <- R - {v}; //Remove v from R.
        } //End foreach loop
     } //End while loop
     //Step 2: Assign vertices into layers.
     L <- {} //Let L be an empty set.
     J <- 1; //Let j be an integer: j = 1.
     while (U not empty) {
        while (U contains a sink) {
          choose a sink u;
          Lᵢ <- Lᵢ  + {u}; //Add u to set Lᵢ.
          U <- U - {u}; //Remove u from U.
          if (|Lᵢ|≥ W)  {   //If Lᵢ has more vertices than W,
              i <-  i + 1; //increment layer number by 1.
            } //End inner while loop.
//If exist v not in layer and S(v) are in layer and |v| is maximized, assign v to layer i.
          if (∀ v | v ∉ L & S(v)∈ L ) {
            Lᵢ <- Lᵢ  + {v}; //Add v to set L.
             U <- U - {v};
           } else {
             i <- i + 1;
           if (|Lᵢ|≥ W)  { //If Lᵢ has more vertices than W,
            i <- i + 1;  //increment layer number by 1.
         }
      }
   }
}
```

Phase 3: Crossing Reduction

This phase reduces the number of edge crossings. As mentioned in Chapter 2, one approach is to perform the *layer-by-layer sweep* algorithm. The *layer-by-layer sweep* algorithm starts from the top and moves through each layer. At each layer the crossing number is computed and is added to a total number of edge crossings. When it reaches the bottom, the algorithm moves upward, again computes the crossing number at each layer, and then adds the crossing number to the total number of edge crossings. Once it reaches the top of the graph, the algorithm compares the previous total edge crossing number against the current total edge crossing number. If the current total edge crossing number is less than the previous result, the algorithm repeats this process until the algorithm no longer finds the total edge crossings are less than the previous runs. Otherwise, the algorithm exits. As discussed in the Barriers and Issues section (page 9), in real-world applications the algorithm can sometimes run a long time before reaching the optimal solution. To accommodate the real-world problem, a maximum allowable iterations value is added into the algorithm as a parameter. The algorithm is terminated when either an optimal solution is found or the specified iterations value is reached.

The pseudocode of the layer-by-layer sweep algorithm is in Table 7.

Table 7. Pseudocode of the layer-by-layer sweep algorithm

```
procedure (G: proper digraph, max: maximum iteration
number)
{
     Let prev_crossings <- 0; //Previous computed numbers
     Let iteration <- 0; //Number of iterations
     Let crossings <- 0; //Number of crossings

     while (iteration < max) {
          foreach ( l_i in G ) {
     //Compute the number of crossing for 2 layered graph.
               crossings <- crossings + barycenter(L_i);
          }
          //Compare against the previous calculation.
          //If crossings is greater than previous number of
          //crossings, the procedure is done.
          if (crossings > prev_crossings ){
             break;
          }
          //If not, continue to calculate the crossing
number
          else {
               prev_crossings = crossings;
          }
          iteration <- iteration + 1;
     }
}
```

To compute the edge crossing number at each layer, a one-sided two-layered

crossing reduction algorithm is performed. As the one-sided two-layered crossing

reduction problem has been studied extensively in the past decade, many algorithms have

been proposed to solve this problem. Results from experiments using real-world graph

data (Marti & Laguna, 2003; Junger & Mutzel, 1997) showed that the barycenter

heuristic often outperforms other algorithms. Hence, the barycenter algorithm will be

employed in the Sugiyama heuristic. The barycenter algorithm is simple and

straightforward. The algorithm first calculates the barycentric value for each vertex u on

layer L_i. It then reorders layer L_i according to barycentric values of vertices. Next, it

calculates the number of edge crossings. The pseudocode of the barycenter algorithm for

the one-sided two-layered crossing reduction problem is displayed in Table 8.

Table 8. Pseudocode of the barycenter algorithm

```
Barycenter procedure (Lᵢ: layer)
{
    Let b(u) be barycentric value of a vertex u in Lᵢ;
    Let u' be vertices on layer Lᵢ₊₁;
    Let Nᵤ be neighbors of u; //In layer Lᵢ₊₁
    pos(u') be position of u' on Lᵢ₊₁;
    foreach (u in Lᵢ ){
```
$$b(u) = \frac{1}{|N_u|}\sum_{u' \in N_u} pos(u'); \text{ //Calculate barycentric value}$$
```
    }
    Sort vertices on layer Lᵢ based on their barycentric
    value;
}
```

Phase 4: Coordinate Assignment

As mentioned in the limitations section of this proposal, the proposed CGDF will

not take into account the actual sizes and shapes of real-world vertices. All vertices will

be circles of the same size. A future framework could take the sizes and shapes of

vertices into account when computing their coordinates.

DynaDAG

The DynaDAG framework (North & Woodhull, 2001) is a dynamic graph

drawing framework that combines both the static layout aesthetic (Sugiyama et al., 1981)

and the dynamic layout aesthetic as factors in drawing algorithms. DynaDAG uses a

client-server model. The client and server exchange messages through *update* operations.

Update operations comprise the following primitive operations: add a vertex, add an edge, remove a vertex and all its incident edges, and remove an edge. Composite updates can be decomposed into those primitive operations. Upon receiving update operations from the client, the server updates an internal model graph by calling a main procedure according to aesthetic criteria for drawing dynamic graph layouts and sends the result back to the client. The client renders the layout to reflect the changes. DynaDAG employs an internal model graph that contains layout and supporting attributes for redrawing the layout due to update operations. This internal model graph satisfies one-level edge constraint for crossing reduction computation (North & Woodhull, 2001). The list of attributes is shown in Table 9. The proposed CGDF will employ a client-server model similar to that of DynaDAG but will use a more complex relational data model. The proposed model graph not only captures vertex and edge attributes and constraints for updating the layouts, but also maintains the snapshots of previous layouts' geometry information. This approach enables clients to render the layout quickly and can provide layout animation if needed. The detailed entity relationship model will be discussed in the section An Entity Relationship for Constrained Hierarchical Graph Drawing.

Table 9. Internal model used in DynaDAG (North & Woodhull, 2001)

Value	Type	Explanation
G = (V;E)	Graph object	Graph
u, v, w… ∈ V	Vertex object	Vertex
e,f, ….∈ E	Edge object	Edge
Δ (G)	Coord	Minimum vertex separation
$L_{i,j}$	Vertex object	j^{th} node in i^{th} layer
R_x, r_y	Float	Precision
$\lambda(v)$	Integer	Layer assignment
X(v), Y(v)	Coord	Position of vertex center
$\hat{X}(v)$ $\hat{Y}(v)$	Coord	Client vertex position
$X'(v), Y'(v)$	Coord	Previous vertex position
b(v)	Coord	Vertex shape bounding box
fixed(v)	Boolean	Node movable
tail(e), head(e)	Vertex object	Endpoints
C(e)	Coord list	Layout spline
$\hat{C}$	Coord list	Client request spline
$\varpi(e)$	Float	Weight > 0
$\delta(e)$	Float	Minimum length > 0
Strong (e)	Boolean	Strong level constraint

The main procedure of DynaDAG is called *Process,* which in common with the Sugiyama heuristic has four phases. Each phase in the Process procedure will examine subgraphs that are affected by update operations, and will update the internal graph according to aesthetic criteria defined in previous sections. The objectives and constraints of each phase in the Process procedure are shown in Table 10. The proposed CGDF will

support operations similar to the insert, update, and delete operations on subgraphs. However, implementation of the CGDF will be different from that of DynaDAG. While DynaDAG transforms Sugiyama phases into optimization problems and uses a network simplex solver for solving those optimization problems, the proposed CGDF will employ modified algorithms that are widely used in each phase of the Sugiyama heuristic for solving these problems.

Table 10. Objectives and constraints of the Process procedure in DynaDAG (North & Woodhull, 2001)

Phase	Objective	Constraint
Phase 1: Remove cycles	Remove cycles	None
Phase 2: Rerank	$\min \sum_{e=(u,v)\in E} w(e)(\lambda(v) - \lambda(u))$	$\lambda(v) \geq \lambda(u) + \delta(u,v)$
Phase 3: ReduceCrossing	Minimize crossings	X(v) = X(u) +1
Phase 4: Coordinate assignment	$\min \sum_{e=(u,v)\in E} w(e)\,\lvert X(v) - X(u) \rvert$	$X(v) \geq X(u) + \Delta(u,v)$

Where:

$w(e)$ is edge weight, which is used as a layout stability constraint.

λ is level or rank assignment.

$\lambda(v)$ is a layer assignment of vertex v.

$\delta(u,v)$ is the minimum length between vertices u and v.

X, Y is the coordinate of a vertex.

Δ is the minimum vertex separation. This value varies, depending on vertex shapes.

In Phase 2, Rerank, DynaDAG transforms the objectives and constraints into optimization problems in which vertices are defined as variables and edges are defined as

constraints, as shown in Tables 11 and 12. North and Woodhull proposed to use an

integer network simplex solver for solving those optimization problems.

Table 11. Variables in Phase 2, Rerank, in the online graph drawing framework (North & Woodhull, 2001)

Variable	Explanation
$\forall v \in V : \lambda(v)$	Layer assignment of v or Y(v)
$\forall v \in V : \tau(v)$	Stable layer assignment of v
$\forall e \in E : \neg strong(e) : \rho(e)$	Lower endpoint of weak edge
$\lambda\min, \lambda\max$	Lowest and highest levels

Table 12. Constraints in Phase 2, Rerank, in DynaDAG (North & Woodhull, 2001)

Constraint Edge	Weight	Explanation
$\forall v \in V : \lambda(v) - \lambda_{\min} \geq 0$	0	Maintain min level
$\forall v \in V : \lambda_{\max} - \lambda(v) \geq 0$	0	X(v) = X(u) +1
$\forall e = (u,v) \in E : strong(e) : \lambda(v) - \lambda(u) \geq \delta(e)$	$\varpi(e)$	Strong edge constraint
$\forall e = (u,v) \in E : \neg strong(e) : \rho(e) - \lambda(u) \geq 0$	$\varpi(e)$	Weak edge constraint
$\rho(e) - \lambda(u) \geq \delta(e)$	$c_{rev}\varpi(e)$	

Where:

c_{rev} is a cost that associates edges.

strong(*e*) is a strong edge constraint.

$\neg strong(e)$ is a weak edge constraint.

North and Woodhull (2001) pointed out that linear network simplex does not take

into account layout stability. To compensate, North and Woodhull added variables and

constraints that penalized the level assignment. In this phase, DynaDAG provides a

tradeoff between geometry stability (global optimization) and minimizing edge length (local optimization) by adjusting the edge constraints.

In Phase 3, ReduceCrossing, DynaDAG does not take into account layout stability. Thus, the crossing reduction problem is solved using a median algorithm without considering the constraints on vertices. Unlike DynaDAG, our proposed algorithm for solving the crossing reduction problem for constrained one-sided two-layered graph layouts will take into account layout stability. Based on the work of Forster (2004), the proposed algorithm will explicitly include a constraint that represents layout stability. The constrained crossing reduction problem and the work of Forster (2004) will be reviewed in the next section.

Phase 4 of the Process procedure, Coordinate assignment, will not be discussed in this dissertation; as mentioned in the Limitations of the Study section in Chapter 1, this dissertation will consider all vertices and edges to have constant sizes and shapes. Hence, coordinate assignment in our proposed constrained hierarchical drawing framework will simply place vertices and edges based on predefined values and their layer assignments.

This section reviews the DynaDAG framework and its main procedure, and discusses the similarities and differences between the DynaDAG framework and the proposed CGDF. Like DynaDAG, the CGDF will use a client-server model for communication between client and server. Another similarity is the transformation of hierarchical graph drawing objectives and aesthetic constraints into equivalent optimization problems. However, the proposed CGDF will use a different approach and

technique in solving the optimization problems. The similarities and differences between

the two frameworks are described in Table 13.

Table 13. Similarities and differences between DynaDAG and the proposed CGDF

| Framework	DynaDAG	CGDF
Data structure	Captures a constraint of one layer assignment	Will store vertex attributes and constraints and also snapshots of previous layouts
Framework	Client-server model	Same as DynaDAG
Heuristic	Based on the Sugiyama heuristic: translates each phase to an optimization problem	Based on both static aesthetic and dynamic aesthetic criteria to develop optimization problems
Phase 1: Cycle Removal	Uses a dot program (previous version of DynaDAG) to solve if applicable	Will use Greedy-Cycle-Removal algorithm (Eades et al., 1993) for removing cycles if applicable
Phase 2: Layer Assignment	Uses integer network simplex. To improve layout stability, adds constraints to penalize the layer assignment.	Will use a modified Coffman-Graham to assign vertices into layers
Phase 3: Crossing Reduction	Employs median algorithm without taking into account layout stability	Modified barycenter algorithm based on the work of Forster (2004)
Phase 4: Coordinate Assignment	Uses network simplex for assigning coordinates	Out of scope of this dissertation
Operations	Adds vertices, adds edges, removes vertices, removes edges	Will add vertices, add edges, remove vertices, remove edges, and add/update vertex constraints

Constrained Crossing Reduction for One-Sided Two-Layered Graph Layouts

     This section first briefly reviews constrained crossing reduction for one-sided two-layered graphs. Second, it reviews the work of Forster (2004) in the constrained crossing reduction problem. Finally, it discusses how that work will be used in our proposed constrained crossing reduction problem solution.

Constrained one-sided two-layered graph layout is a variant of one-sided two-layered graph layout in which some pairs of vertices are restrained from changing sequence. For example, given $u, v \in L_i : c(u, v)$ is constrained *iff* $pos_i(u) < pos_i(v)$. A constraint between vertices u and v is be denoted as $c(u, v)$. Algorithms for solving constrained one-sided two-layered graphs must take into account these constraints while reordering the vertices on layers. The constraint is satisfied only if $pos(u) > pos(v)$ (Forster 2004).

Forster (2004) proposed a simple solution for solving crossing reduction for constrained one-sided two-layered graphs ($G = (V, L_i, L_{i+1})$), based on the barycenter algorithm. The main idea of the algorithm is based on the following observations:

1. The barycenter algorithm sorts the vertices on L_i based on each vertex's barycentric value from left to right, so a vertex with a greater barycentric value will be placed on the right and the vertex with the lesser barycentric will be placed on the left. A constraint $c(u, v)$ on layer L_i is violated if the barycentric value of vertex u is greater than that of vertex v ($b(u) > b(v)$).

2. It is also known that the greater barycentric value of vertex u indicates more edges are to the right of the vertex than to its left. In the same manner, the lesser barycentric value of vertex v indicates more edges are to the left of the vertex than to its right, as shown in Figure 11.
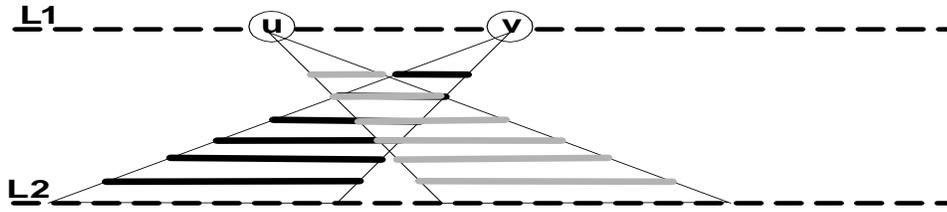
Figure 11. Barycentrics of vertices and their incident edges

Based on this observation, Forster (2004) proposed a simple solution. To minimize the number of edge crossings without violating the constraints, no other vertices should be placed between u and v. Forster noted that although this assumption is not true in general, the experimental result shows that the assumption produces good layouts. The modified barycenter for constrained one-sided two-layered graphs proposed by Forster is described as follows:

Given that $(G, L_i, L_{i+1}) \mid L_{i+1}$ is fixed. To minimize the number of crossings in layer L_i, the algorithm first calculates the barycentric values of the vertices. Second, it partitions the vertices into total order vertex lists with one singleton list per vertex. Third, the algorithm loops through the vertices to find violated constraints. For each constraint $c(u, v)$ that is violated, a dummy vertex is created. This dummy vertex is used as a surrogate for both vertices u and v, and the barycentric value of the dummy vertex is the average value of the barycentric of vertices u and v. This step ensures that when sorting vertices on L_i no vertices are placed between the vertices u and v, as both u and v are now replaced by a single dummy vertex. Next, the algorithm sorts vertices based on their barycentric values. Finally, the algorithm replaces all dummy vertices with the original vertices u and v. Table 14 shows the pseudocode of the modified barycenter algorithm, and Table 15 shows the pseudocode of the algorithm that finds violated constraints.

Table 14. Pseudocode of the modified barycenter algorithm (Forster, 2004)

```
Barycenter procedure (G=(L₁, L₂, E): two-layered graph, C=( L₂xL₂:
acyclic constraints)
{
    Let b(u) be the barycentric value of a vertex u in Lᵢ;
    Let u′ be vertices on layer Lᵢ₊₁;
    Let Nᵤ be neighbors of u; //In layer Lᵢ₊₁
    Let pos(u′) be the position of u′ on L₂;
    Let V be a set of vertices that have constraints;
    Let V′ be a set of vertices that do not have constraints;
    L(u) <- {}; //Empty list
    foreach (u in L₂ ) { //Calculate barycentric values.
```

$$b(u) = \frac{1}{|N_u|} \sum_{u' \in N_u} pos(u'); \text{ //Barycentric value of } u$$

```
        L(u) <- (u) //New singleton list
    }
    V <- {u,v | (u,v) ∈ C}; //Add (u, v) to V.
    V′ <- L₂ - V;  //Vertices that do not have constraints

    while((u,v) <- find_violate_constraint(V,C) <> empty) {
        create new vertex v_c; //Dummy vertex
        deg(v_c) <- deg(u) + deg(v);
```

$$b(v_c) <- \frac{(b(u)*\deg(u)+b(v)*\deg(v))}{\deg(v_c)};$$

```
        L(v_c) <- L(u) × L(v);
        for (c ∈ C) { //Add incident edges of u or v to v_c.
          if (c is incident to u or v) {
            make c incident to v_c;
          }
        }
        C <- C - {(v_c, v_c)}; //Remove self loop.
        V <- V - {u, v}; //Remove (u,v) from V.
        if (v_c has incident constraint) {
          V <- V ∪ {v_c}; //Add v_c into V.
        }
        else {
          V′ <- V′ ∪ {v_c}; //Otherwise, add v_c into V′·
        }

        V′′ <- V ∪ V′; //Union V and V′
        Sort V′′ by b(); //Sort V′′ by barycentric values.
        L <- {}; //Empty the temporary list.
      //Concatenate and replace dummy vertices by the
      //original vertices.
        for (v ∈ V′′) {
          L <- L × L(v);  //Concatenate vertices into a list.
          }
    }
    return L;
}
```

Table 15. Pseudocode of the algorithm that finds violated constraints (Forster, 2004)

```
Find_violated_constraints procedure (V: set of violated vertices, C:
constraints)
{
    S <- {};   //Let S be an empty list.
    for (each v in L₂) {
       I(v) <- {}; //Let I be empty list.
       if (indeg(v) = 0) { //New singleton list
         S <- S ∪ {v};     //Vertices without incoming edges
       }
    }

    while(S <> {}) { //While S is not empty
       select v ∈ S;
       S <- S - {v};
       foreach (c =(u,v) ∈ I(v)) {
         if (b(u) ≥ b(v)) {
           return c;
         }
       }

       foreach (c = (v,t)) {
        I(t) <- {c} × I(t);
        if (|I(t)| = indeg(t)) {
           S <- S ∪ {t};
        }
       }
    }
    return null;
}
```

In this section we reviewed the Online Graph Drawing framework (North & Woodhull, 2001), the relationship between local drawing quality and global layout stability (Görg, 2005), the standard Sugiyama heuristic for drawing hierarchical graphs, and a fast heuristic for constrained one-sided two-layered graphs proposed by Forster (2004). The combination of North and Woodhull's (2001) Online Graph Drawing framework and the relationship between local drawing quality and global layout stability (Görg, 2005) will influence the design of an abstract formal model for constrained graph layout. The aesthetic criteria for drawing incremental hierarchical graph layouts help to
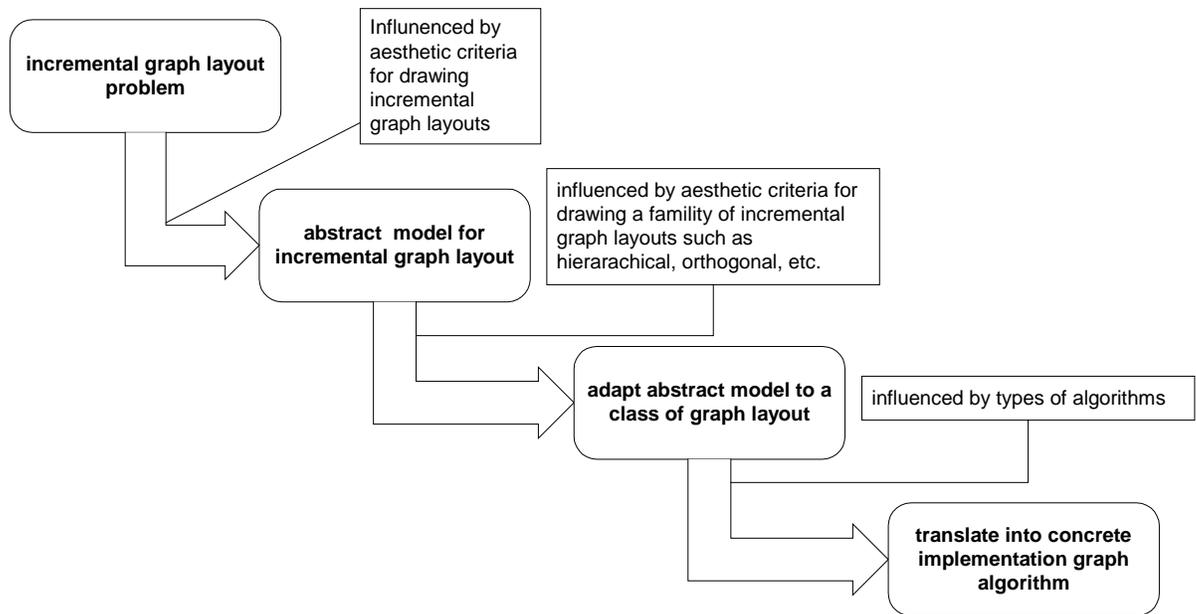
build an abstract model for drawing comprehensible hierarchical graph layouts, and the standard Sugiyama heuristic provides a foundation for developing concrete algorithms for updating the constrained graph layouts incrementally as end users update the graphs.

Proposed Constrained Incremental Graph Drawing Framework

This section presents a constrained incremental graph drawing framework. First, it discusses a simple approach to designing an abstract model for drawing incremental graph layouts. Next, it gives details of that design. Third, it discusses a model for drawing hierarchical graph layouts. Next, it presents a mapping of the proposed abstract model into concrete algorithms based on aesthetic criteria and the Sugiyama heuristic. Finally, it presents pseudocode for modified Sugiyama algorithms for drawing constrained graph layouts.

Design of an Abstract Model for Incremental Graph Layouts

Designing an abstract model for incremental graph layouts uses a top-down approach, as shown in Figure 12. First, an abstract model is designed that represents incremental graph layout regardless of the family of graph layouts. Next, the abstract model is adapted to represent a family of graph layouts such as hierarchical graph layout. Finally, the abstract model is transformed into concrete algorithms. This approach enables future research to extend the work of the dissertation by developing models for other types of graph layouts such as orthogonal, simulated annealing, etc.

incremental graph layout problem

Influenced by aesthetic criteria for drawing incremental graph layouts

abstract model for incremental graph layout

influenced by aesthetic criteria for drawing a famility of incremental graph layouts such as hierarachical, orthogonal, etc.

adapt abstract model to a class of graph layout

influenced by types of algorithms

translate into concrete implementation graph algorithm

Figure 12. Design flow for building an abstract model for incremental graph layouts

Details of the Abstract Model

As discussed in the earlier section Aesthetic Criteria for Incremental Graph

Layouts, the three important aesthetic criteria are consistency, layout stability, and

readability. The proposed abstract model for incremental graph layout is based on the

work of North and Woodhull (2001), which is an optimization problem of aesthetic

criteria. According to North (1995), consistency has the highest level of importance

because it maintains the characteristics of the type of graph being created. Layout

stability is purely heuristic (North, 1995). To formalize the importance of each constraint,

our proposed abstract model relies on recent research by Huang and Eades (2005) and

Görg (2005).

Huang and Eades (2005) performed an experiment on how humans read graphs.

The result showed that reducing the number of edge crossings without allowing user

constraints on layout stability may not improve readability in all cases. Furthermore,

Görg (2005) showed that readability or local layout quality conflicts with layout stability

or global layout quality. Hence, in our proposed abstract model the values of weights of

drawing readability and layout stability shall be defined by end users, and the optimal

solution for this problem will be a balance between *local drawing quality* and *global*

drawing quality. In summary, the proposed abstract model for incremental graph layout is

an optimization problem of three aesthetic criteria. Each criterion is weighted by its

importance, as shown in optimization problem (1), where Θ is the optimization goal of

the proposed abstract model, C is consistency, R is readability, and S is stability, where

w_c, w_r, and w_s are weights of consistency, readability, and stability respectively.

$$
\begin{cases}
\Theta = w_c C + w_s R + w_r S \\
\text{maximize } \Theta \\
w_c, w_s, w_r \geq 0 \\
w_c + w_s + w_r = 1 \\
w_c > w_r \\
w_c > w_s
\end{cases}
\quad \textbf{(1)}
$$

This proposed abstract model depends only on aesthetic constraints, not on the

types of metrics that measure layout stability (orthogonal, near neighbor, etc.) or the

types of algorithms for drawing graph layouts (force-directed, hierarchical, or orthogonal

layout). Both consistency and readability constraints are embedded within standard

algorithms for drawing layouts. The layout stability constraint can be calculated using

appropriate measuring metrics and can also be adjusted by end users. Thus, the proposed

abstract model can be adapted to different types of graph layouts without affecting the

actual implementation of the algorithm or the type of layout. Based on the proposed abstract model we can define a constrained graph layout as follows:

Given a sequence of n graphs $g_1, g_2, ..., g_n$. Compute layouts $l_1, l_2, ..., l_n$ for these graphs such that Θ is optimal, where Θ is an objective function of three aesthetic criteria. This definition can be applied to either online or off-line dynamic graph layouts.

An Abstract Model for Hierarchical Constrained Graph Layouts

In the preceding sections, a generic abstract model for incremental graph layouts was introduced. This section extends the proposed abstract model for incremental graph layouts to develop an abstract model for incremental hierarchical graph layouts. Unlike algorithms for the directed force layout model and orthogonal graph layouts, hierarchical graph drawing algorithms like the Sugiyama heuristic are *multi-phased*. North and Woodhull (2001) observed that there is no unified model that represents hierarchical graph drawing algorithms, but the aesthetic criteria should be divided to form different constraints in each phase of the Sugiyama algorithm. Görg et al. (2004) addressed the same issue with the hierarchical graph layout family when implementing the Foresighted Layout algorithm for drawing dynamic hierarchical graphs. They noted that no global graph adjustment for hierarchical graph layouts exists. Hence, Görg et al. (2004) divided the adjustments into multiple steps in accordance with the Sugiyama heuristic. Similar to the model for the One Graph Drawing framework (North & Woodhull, 2001), the proposed abstract model for hierarchical graph layout will comprise suboptimization problems corresponding to each phase in the Sugiyama heuristic.

As discussed in Chapter 2, Sugiyama has four phases, so the proposed abstract model for hierarchical incremental graph layout will include four suboptimization problems if applicable. The generic abstract model (Equation 1) introduced in the previous section shall be used as the foundation for each suboptimization problem.

The first phase in the Sugiyama heuristic, which temporarily reverses the directions of edges, affects none of the aesthetic criteria so the weights of all three aesthetic constraints in the optimization problem for this step are set to 0. Moreover, because the proposed hierarchical constrained graph drawing system will use a relational data model to capture the graph structure, which includes edge direction, any set of edges that needs to be reversed will be identified automatically once the graph model is built. Thus, the first phase of the Sugiyama heuristic will be solved using prior knowledge stored in a relational database and that step will involve no optimization problem.

The second phase, layer assignment, which assigns vertices into layers, not only alters the positions of vertices in the layer but also potentially moves vertices from one layer to another. This step does affect all three aesthetic criteria. Hence, the optimization problem for layer assignment involves all three constraints. The abstract model for the second step will be same as the abstract generic model:

$$
\begin{cases}
\Theta = w_c C + w_s R + w_r S \\
\text{maximize } \Theta \\
w_c, w_s, w_r \geq 0 \\
w_c + w_s + w_r = 1 \\
w_c > w_r \\
w_c > w_s
\end{cases}
\quad \textbf{(2)}
$$

The third phase in the Sugiyama algorithm, crossing reduction, minimizes the number of crossings by reordering vertices on a layer. This phase affects readability and layout stability but not consistency, because this phase does not change the orientation of vertices or alter the characteristics of the hierarchical graph layout. Hence, the weight of the consistency constraint is set to 0. The optimization problem for the crossing reduction problem comprises readability and layout stability constraints, as shown in Equation 3. The abstract model for the third phase will be as follows:

$$\begin{cases} \Theta = w_s R + w_r S \\ \text{maximize } \Theta \\ w_r, w_s \geq 0 \\ w_s + w_r = 1 \end{cases} \quad \textbf{(3)}$$

Though the fourth phase, coordinate assignment, does impact readability and layout stability, the scope of this research does not include shapes and sizes of vertices, as mentioned in the Limitations of the Study section, Chapter 1, page 11. Coordinate assignment in the proposed incremental graph drawing framework is simply a constant function. A future improvement of this dissertation will take into account the different sizes and shapes of vertices.

The Modified Sugiyama Heuristic for Constrained Incremental Graph Layouts

This section translates the optimization problems for drawing incremental hierarchical graph layouts presented in the previous section to appropriate algorithms in the Sugiyama heuristic. Our proposed solution for preserving global layout stability is slightly different from those of both off-line (North & Woodhull, 2001) and online (Görg, 2005) approaches. DynaDAG (North & Woodhull, 2001) used a network simplex solver

to solve optimization problems. To preserve layout stability, additional constraints are added to the linear optimization problems. DynaDAG preserves layout stability by basing each layout solely on the previous layout, but according to Görg (2005), that approach may require redrawing the entire graph layout. Görg proposed to use a global graph layout configuration to preserve layout stability, an improvement on the online graph layout framework. However, Görg noted that this approach does not work automatically with multi-phase algorithms like hierarchical graph layout algorithms. To accommodate the Sugiyama heuristic, Görg divides the global adjustments into multiple phases in order. The proposed constrained incremental graph drawing framework will use a modified Sugiyama heuristic to update the graph layouts. Because none of the algorithms in the standard Sugiyama heuristic take into account layout stability, we propose to preserve stability by embedding a simple solution within the Sugiyama algorithms.

To preserve layout stability and to ease its incorporation into a multi-phase algorithm like the Sugiyama heuristic, the proposed CGDF introduces two new vertex attributes. The first attribute, called *k*, preserves layout stability while reassigning vertices to layers (Step 2 in Sugiyama) by restricting a vertex from changing layers. The second attribute, *c(u, v)*, also preserves layout stability; it minimizes the crossing numbers for one-sided two-layered graph layouts by restricting vertices from changing their sequence within a layer.

Attribute *k* of a vertex is defined as $k \in [0,1]$, where *k = 0* means a vertex is not constrained in its movement and *k =1* means the vertex cannot be moved. Initially, *k* is set to zero for all newly created vertices. To inhibit vertices from moving away from their

original positions, the value of k is increased if a vertex is moved away from its original position and is decreased if the vertex is moved closer to its original position. Accommodating the Huang and Eades' (2005) experiment, end users can adjust the value of k as needed to balance global layout stability with drawing readability. The k attribute is stored in a relational database with other vertex attributes.

The attribute $c(u, v)$ of a vertex is defined as $c(u, v) \in [0,1]$, where u, v $\in V$ and u and v are in the same layout. The attribute $c(u, v)$ represents an order constraint between vertices u and v on a layer. Initially, all vertices have the value of $c(u, v)$ set to zero. To ensure that this attribute preserves layout stability but does not become so restrictive that it degrades drawing readability, the proposed framework enables end users to adjust $c(u, v)$ at run time. This attribute is also stored in the relational database.

Step 2, layer assignment, as discussed in the previous section, does impact all three aesthetic criteria as shown in Equation 2. Hence, the layer assignment algorithm should take into account all three criteria while reassigning vertices, which are affected by update operations, to layers. We will embed both consistency and readability criteria in the layer assignment algorithm. For instance, assigning vertices to layers and pointing their edges in the same direction satisfy the consistency criterion for a hierarchical graph layout. Keeping the width and height of the layout proportional satisfies the readability criterion. To accommodate layout stability, the modified layout assignment algorithm for the proposed framework will use the newly introduced attribute k, whose value will be used to determine whether a vertex will be assigned to a new layer.

The modified version of the Coffman-Graham algorithm will be similar to the original algorithm but will accept a sub-graph instead of the entire graph model. First, the algorithm loops through the sub-graph and computes the lexicographical order for each vertex. In Step 2, vertices are sorted based on their labels. In Step 3, assigning vertices into layers, the algorithm uses k to determine whether a vertex can be assigned to a new layer. For each vertex, if its k value equals 1 the vertex will remain in the same layer as it was in the previous layout. Otherwise, the vertex will be placed in a new layer and the k value will be increased. Table 16 shows the pseudocode of the modified layer assignment algorithm as applied to each vertex.

Table 16. Pseudocode of the modified Coffman-Graham algorithm

```
procedure (G: reduced graph, W:positive integer, delta: real number)
{
    Let Π : v → {0,n} be integer label of v
    Let N(v) be a lexicographical order of v
    Let P(v) be predecessors of v
    Let S(v) be successors of u
    Let R(v) be a set of v
    Let i be an integer: i <- 0
    Let U <- {} be an empty set
    Let L_prev(v) be the previous layout of v
    Let k(v) be constraint movement of v
    //initialize all k(u) to zero
    k(u) ← 0
    //Step 1: assign integer labels to vertices. All vertices are
    //unlabelled.
        for (each u in G) {
           if (P(u) == 0){ //Select all the sources and isolated
                           //vertices.
            i <- i + 1; //Increment i by 1.
            Π(u) <- i;  //Assign an integer label i to u.
            U <- U + {u}; //Add u to U.
            G <- G - {u}; //Remove u from G.
          }
       }
      while (G not empty) {
        //Choose set R(v) : unlabelled v that has no unlabelled
predecesors.
        foreach ( v in G) {
          if (Π(v) == 0 && P(v) are labelled) {
             R <- R + {v}; //Add v to R.
```

```
            G <- G - {v}; //Remove v from G.
          }
        } //End foreach loop
        //Sort the set R based on lexicographical order of the P(v)
labels
        //Assign label i to vertices and add them to the set U.
        foreach ( v in R) {
          i <- i + 1;
          Π(u) <- i;
          U <- U + {v}; //Add v to U.
          R <- R - {v}; //Remove v from R.
        } //End foreach loop.
      } //End while loop.
```

```
//Step 2: Assign vertices into layers.
      L <- {}; //Let L be an empty set.
      J <- 1; //Let j be an integer: j = 1.
      while (U not empty) {
         while (U contains a sink) {
           choose a sink u;
           if (k(u) == 1) {
             L_prev(u) <- L_prev(u) + {u}; //Add u to the same layer it was.
           } else {
             L_i <- L_i + {u}; //Add u to set L_i.
             if (L_prev(u) <> L_i) {
               k(u) <- k(u)+ delta; //Increment value k
           }
           U <- U - {u}; //Remove u from U.
           if (|L_i|≥ W)  {    //If L_i has more vertices than W,
             i <-  i + 1; //increment layer number by 1.
           }  //End if statement.
        } //End inner while loop.
           //If exist v is not in layer and S(v) are in layer and |v| is
           //maximized, assign v to layer i.
           if (∀ v | v ∉ L & S(v) ∈ L ) {
             if (v_k == 1) {
               L_prev(v) <- L_prev(v)+ {v}; //Add v to the same layer it was.
             } else {
               L_i <- L_i  + {u}; //Add u to set L_i.
               if (L_prev(v) <> L_i) {
                 k(v) <- k(v) + delta; //Increment value k.
               }  //End inner if statement.
               U <- U - {v}; //Remove v from U.
            } else {
              i <- i + 1;
            }
            if (|L_i|≥ W) { //If L_i has more vertices than W,
              i <- i + 1;  //increment layer number by 1.
            }
      } //end while loop
}
```

For Phase 3, crossing reduction, the proposed constrained hierarchical graph drawing system will use a modified barycenter algorithm for solving constrained crossing reduction, which extends the work of Forster (2004). The original barycenter algorithm was not designed to preserve layout stability, so the modified barycenter algorithm will employ the attribute $c(u, v)$ to determine whether the order of vertices on a layer can be changed.

The modified one-sided crossing reduction algorithm comprises two steps. The first step is to calculate barycentric values for the vertices. The second is to resequence vertices on the layers based on their barycentric values.

In the first step, the modified algorithm will use a vertex's information stored in a relational database to determine whether to compute a barycentric value for the vertex. We observe that the barycentric value of a vertex v on a layer L_i depends only on the number of its neighboring vertices on layer L_{i+1}. For each vertex, the modified barycenter algorithm will compare the delta between the previous outdgree value stored in the relational database and the current outdgree value of that vertex. If the delta is not 0 the algorithm will compute the barycentric value for that vertex. Otherwise, the algorithm will proceed to the next vertex. The second step is to reorder vertices on a layer based on their barycentric values. If a position order constraint $c(u, v)$ between vertices u and v is violated *(b(u) > bv)),* the algorithm shall employ a procedure similar to the one proposed by Forster (2004) to reorder the vertices on a layer without violating *c(u, v).* Table 17 shows the pseudocode of the modified barycenter algorithm. Table 18 shows the differences between the proposed algorithm and the work of Forster (2004).

Table 17. Modified Barycenter algorithm for the one-sided two-layered constrained crossing reduction problem

```
Barycenter procedure (G=(L₁, L₂, E): two-layered graph, C=( L₂xL₂: acyclic
constraints)
{
    Let b(u) be the barycentric value of a vertex u in Lᵢ
    Let u' be vertices on layer Lᵢ₊₁
    Let Nu be neighbors of u; //In layer Lᵢ₊₁
    Let Nuₚᵣₑᵥ be neighbors of u in previous layout
    pos(u') be the position of u' on L2
    Let V be a set of vertices that have constraints
    Let V' be a set of vertices that do not have constraints
    c(u,v) ordered constraint of vertices u and v
    L(u) <- {}; //Empty list
    foreach (u in L₂ ) { //Calculate barycentric values.
       if (Nuₚᵣₑᵥ <> Nu ) {
```

$$b(u) = \frac{1}{|N_u|} \sum_{u' \in N_u} pos(u') \; ; \quad //\text{Barycentric value of u}$$

```
       } //End if statement.
       L(u) <- (u) //New singleton list
    } //End for loop.
    V <- {u,v | (u,v) ∈ C}; //Add (u, v) to V.
    V' <- L₂ - V;  //Vertices that do not have constraints
    foreach (u in L₂) {
      //retrieve c(u,v) from relational database
      // findOrderedConstraint = access database to retrieve c(u,v)
      c(u,v) = findOrderedConstraint(u,v);
      // If constraint is < 1, create a dummy Vc
      if ( c(u,v) <> NULL && b (v) > b(u)) {
           create new vertex vc; //Dummy vertex
           deg(vc) <- deg(u) + deg(v);
```

$$b(v_c) \leftarrow \frac{(b(u) \times \deg(u) + b(v) \times \deg(v))}{\deg(v_c)} \; ;$$

```
           L(vc) <- L(u) ✕ L(v);
           for (c ∈ C) { //Add incident edges of u or v to vc
             if (c is incident to u or v) {
               make c incident to vc;
             } //End if statement.
           } //End for loop.
           C <- C - {(vc, vc)}; //Remove self loop.
           V <- V - {u, v}; //Remove (u,v) from V
      } //end if statement
      if (vc has incident constraint) {
        V <- V ∪ {vc}; //Add vc into V.
      } else {
        V' <- V' ∪ {vc}; //Otherwise, add vc into V'
      } //End if else statement.
      V'' <- V ∪ V'; //Union V and V'
      Sort V'' by b(); //Sort V'' by barycentric values
      L <- {}; //Empty the temporary list.
     //Concatenate and replace dummy vertices by the original vertices
      for (v ∈ V'') {
          L <- L ✕ L(v);  //Concatenate vertices into a list.
      }  //End for loop.
```

```
    } //End outer for loop.
    return L;
}
```

Table 18. Differences between the proposed algorithm and the work of Forster (2004)

Proposed Algorithm (CGDF)	Forster (2004)
Compute barycentric values for vertices that have different neighborhoods.	Compute barycentric values for all vertices.
Need not find vertices that have violated constraints.	Find vertices that have violated constraints.

Architecture of the Proposed Constrained Graph Drawing Framework

The proposed CGDF architecture will be similar to that of the DynaDAG (North & Woodhull, 2001). It will be a client-server application. The client will include a drawing editing tool and a visualization module. The editing tool will be a command line style program that will enable end users to update graph layouts by either entering a set of instructions into a console or loading a file that contains a series of instructions into the program. A simple language for graph editing will also be defined to provide a set of instructions for updating graph layouts. A visualization module is a simple program that displays graph layouts. This program will be able to run either on a desktop or online. Similar to DynaDAG, the CGDF will support five basic operations:

1. Add vertices

2. Add edges

3. Remove a vertex and all of its incident edges

4. Remove edges

5. Add constraints to vertices

6. Remove constraints from vertices

The CGDF will use TCP and HTTP protocols for communication between clients and a server. Once a user executes an instruction using the editing tool, the client will send the command to the server. The server will then update the graph layout based on the *update* operations and store a new snapshot of the layout in a relational database.

Entity Relationship Diagram for Constrained Hierarchical Graph Drawing

To render graphs with thousands of vertices, the proposed constrained hierarchical graph drawing framework will use a relational data model to capture the graph model, and a sequence of graph layouts to speed up graph drawing and visualization. Computing hardware has been progressing rapidly in the past decade, especially in the data storage field, so our design will take advantage of this. In interactive graph drawing and visualization applications, especially Internet applications, performance in rendering graph layouts has a higher priority than reducing graph data storage space. Thus, a relational data model will be utilized to contain snapshots of each layout in the graph layout sequence. The entity relationship diagram for the model is shown in Figure 13.
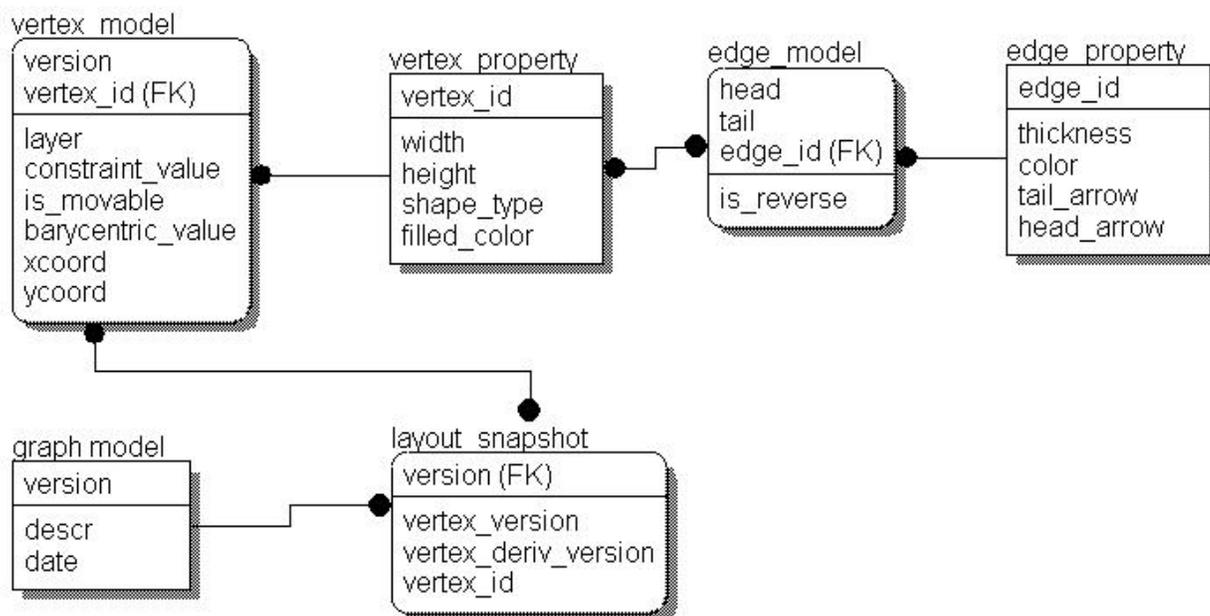
Figure 13. Entity relationship diagram for constrained hierarchical graph layouts

The table vertex_model will store vertices from the latest version of the

hierarchical graph layout. The table will contain necessary vertex properties that will be

used in the proposed modified Sugiyama heuristic. Though the work of this dissertation

will not take into account the sizes and shapes of vertices, the model will be designed to

include physical properties of vertices such as width, height, color, and shape types. The

model will enable the framework to be extended in future work without changing its

foundation or the data structure. The physical properties of vertices will be stored in a

table called vertex_property. The vertex_model table will associate with this property

table through the primary key vertex_id. In the same manner, the edge_property table

will store characteristics of an edge and associate with the edge_model table, which will

represent edges in the hierarchical graph layout. The graph_model table represents the

current state of the graph and the layout_snapshot table represents a layout. To minimize

data storage and avoid redundancy, an attribute called vertex_deriv_version will be used.

This attribute will represent a logical pointer to an actual vertex by associating with the attribute vertex_version.

The Process of Collecting Graph Data

We will use synthetic graph data and data in the public domain. We will collect public graph data from the following sources:

- The Stanford GraphBase

- The US National Institute of Standards and Technology (NIST)

- The Atlas of Cyberspaces

- The Cobot project

Testing and Evaluation

We will employ suggestions by Eades (2005) to evaluate and measure our proposed hierarchical graph drawing framework and its constrained crossing reduction algorithm. The proposed framework and algorithm will be evaluated and measured based on the three goals shown in Figure 14. Effectiveness will measure how well the framework produces layouts. Efficiency will measure the performance of the algorithm and the scalability of the proposed framework, such as how well the system handles large graph layouts. Elegance will measure the extensibility of the graph drawing framework and how easily it can be adapted to other classes of graph layouts.
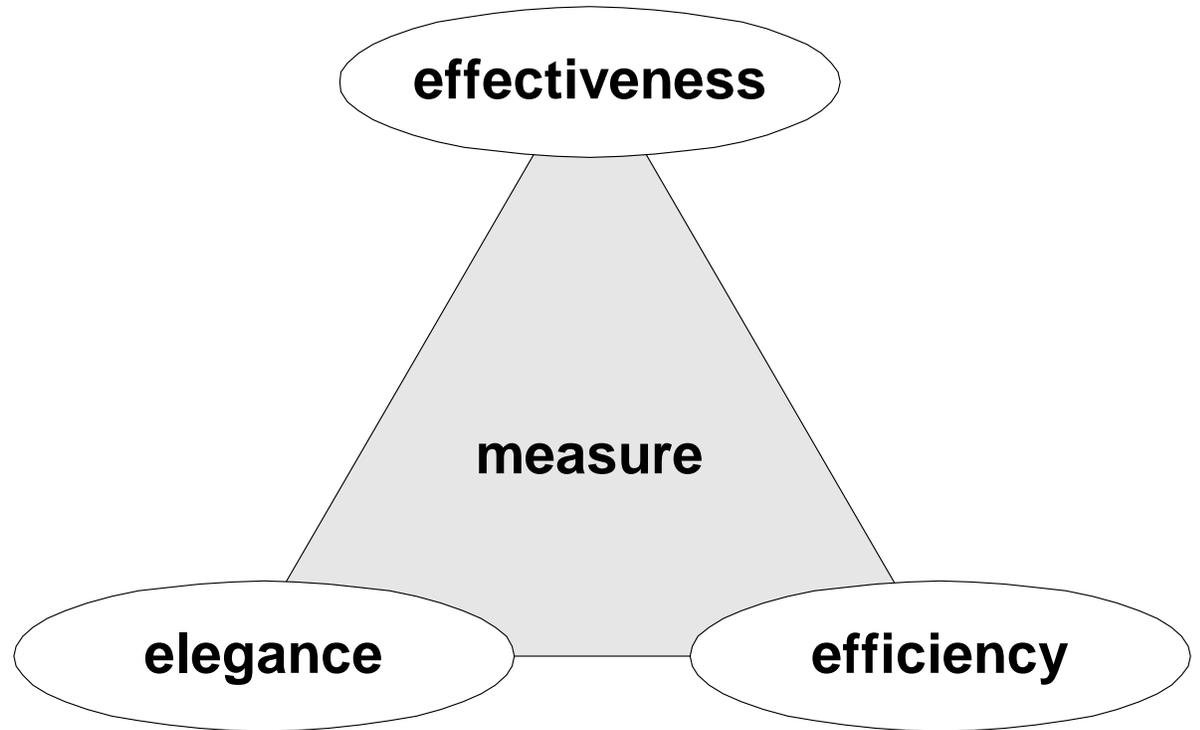
Figure 14. Measurable goals for evaluating the proposed algorithm

To measure the effectiveness of our proposed framework, we will produce a test using well defined graph layouts that have been used in the past to measure the effectiveness of standard graph drawing algorithms. The result should look similar to the expected layouts.

To measure the efficiency of our proposed framework and algorithm, we will conduct several tests. One test will measure how well our graph drawing framework handles a graph data set with a few thousand vertices. The rendering time should be within 15 seconds. The graph data for this test will be from randomly generated synthesized graph data and graph data from the aforementioned public domains. The second test will compare the proposed framework with DynaGraph (North & Woodhull,

2001). The result should be that the proposed framework performs either better than or comparable to the DynaGraph framework. The third test will measure the performance of our constrained crossing reduction against the algorithm proposed by Forster (2004).

We will leave elegance to the future recommendations section of the dissertation, because to extend the proposed abstract model to another class of graph layout is out of scope for the proposed dissertation.

Resource Requirements

The following resources will be used in this research:

- Hardware: A desktop computer to serve as a web server that hosts the graph visualization application, and a database server that stores the graph model. Laptops will be used as clients connected to the graph framework.

- Software: Java Development Kit (JDK), Apache Tomcat Servlet engine, MySQL relational database server, Java open source graph libraries.

- Graph data: NIST, Stanford GraphBase, Atlas of Cyberspaces, Cobot project, http://www.graphdrawing.org/, and synthetic graph data.

- Participant: The author will be the only researcher for this project.

Summary

This chapter first discusses aesthetic criteria for drawing hierarchical graph layouts and additional aesthetic criteria for incremental graph layouts. Next, we review the Sugiyama heuristic in detail because the proposed hierarchical graph drawing framework will use the standard Sugiyama heuristic in building the initial graph model from an initial data set. The online dynamic graph drawing framework (North &

Woodhull, 2001) is then reviewed in detail, as it will be a foundation for the proposed constrained hierarchical graph drawing framework. The constrained crossing reduction problem is then reviewed, as we will extend the work of Forster (2004) to solve that problem.

We then present an abstract optimization model based on dynamic aesthetic criteria; we will adapt the proposed model for hierarchical graph layout. The optimization problem is then incorporated into appropriate steps in the Sugiyama heuristic. The pseudocode of the modified Sugiyama heuristic that will be used to solve those optimization problems are then presented. We also present an overview architecture of the proposed constrained graph drawing framework and six basic graph editing operations. An entity relationship model that will be used to capture the layout snapshots is presented next. Finally, we present measurable goals for testing and evaluating our proposed graph drawing framework and algorithm.

Reference List

Battista, G. D., Eades, P., Tamassia, R., & Tollis, I. (1999). *Graph drawing algorithms for the visualization of graphs*. New Jersey: Prentice Hall.

Bohringer, K., & Newbery, P. (1990). Using constraints to achieve stability in automatic graph layout algorithms. *Proceedings of ACM CH 90,* 43-51.

Brandes, U., & Wagner, D. (1997). A Bayesian paradigm for dynamic graph layout. *Proc. Measures for GSymp. Graph Drawing GD '97,* pp. 236-247.

Bridgeman, S., & Tamassia, R. (2002). A user study in similarity measures for graph drawing. *Journal of Graph Algorithms and Applications, 6*(3), 225-254.

Buchsbaum, A. L., & Westbrook, J. R. (2000). Maintaining hierarchical graph views. *Proceedings of the Eleventh Annual ACM-Siam Symposium on Discrete Algorithms*. 566-575.

Catarci, T. (1988). The assignment heuristic for crossing reduction. *IEEE Trans. Syst. Man Cybern. 25*(3), 515-521.

Coffman, E. G., & Graham, R. L. (1972). Optimal scheduling for two-processor systems. *Acta Informica 1*(3), pp. 200-213.

Cohen, R. F., Battista, G. D., Tamassia, R., Tollis, I. G., & Bertolazzi, P. (1992). A framework for dynamic graph drawing. *Annual Symposium on Computational Geometry: Proceedings of the Eighth Annual Symposium on Computational Geometry* (pp. 261-270). Berlin: ACM.

Demetrescu, C., & Finocchi, I. (2003). Combinatorial algorithms for feedback problems in directed graphs. *Information Processing Letters, 86*(3), 129-136.

Davison, R., & Harel, D. (1996). Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics 15*(4), 301-331.

Diehl, S., & Görg, C. (2002). Graphs, they are changing. *Lecture Notes in Computer Science Vol. 2528* (23-30).

Diehl, S., Görg, C., Kerren, A. (2000). Foresighted graph layout. *Technical Report*, FR Informatik, Saarland University.

Eades, P. (2005). *How to get a PhD in Information Technology*. Retrieved May 02, 2007 from http://www.cs.usyd.edu.au/~peter/howtogetphdusyd.pps

Eades, P., & Kelly, D. (1984). The Marey graph animation tool demo. *Proceedings of the 8th International Symposium on Graph Drawing,* 396 - 406.

Eades, P., & Kelly, D. (1986). Heuristics for reducing crossings in 2-layered networks. Ars Combinatoria, pp. 187-191. *Proceedings of the Australian Computer Science Conference*, 327-334.

Eades, P., Lin, X. Y, & Smith, W. (1993). A fast effective heuristic for the feedback arc set problem. *Information Processing Letters, 47,* 12-15.

Finocchi, I. (2002). *Hierarchical decompositions for visualizing large graphs*. Ph. D thesis. Universita̍ degli Studi di Roma, Rome.

Forster, M. (2004). A fast and simple heuristic for constrained two-layered crossing reduction. *Proc. 13th Int. Symposium of Graph Drawing. Lecture Notes in Computer Science 3383,* pp. 206–216.

Frishman, Y., & Tal, A. (2007). On-line dynamic graph drawing. In K. Museth, T. Möller, & A. Ynnerman (Eds.), *Eurographic/ IEEE-VGTC Symposium on Visualization*.

Gansner, E. R., North, S. C., & Vo, K. P. (1993). A technique for drawing directed graphs. *IEEE Transactions on Software Engineering, 19*(3), 214-230.

Garey, M. R., & Johnson, D. S. (1979). *Computers and intractability: A guide to the theory of NP-complete.* New York: W. H. Freeman.

Garey, M. R., & Johnson, D. S. (1983). Crossing number is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, *4(3),* 312-316.

Görg, C. (2005). *Offline drawing of dynamic graphs*. Ph. D Dissertation. *Saarland University, Saarbrücken, Germany*.

Görg, C., Birke, P., Pohl, M., & Diehl, S. (2004). Dynamic graph drawing of sequences of orthogonal and hierarchical graphs. *Proceedings of 12th International Symposium on Graph Drawing*.

He, W., & Marriott, K. (1998). Constrained graph layout. *Constraints: An International Journal, 3,* 289-314.

Huang, W., & Eades, P. (2005). How people read graphs. *Proceedings of the 2005 Asia-Pacific Symposium on Information Visualisation, Australia, 45,* 51-58.

Junger, M., & Mutzel, P. (1997). 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *J. Graph Algorithms and Applications, 1*(1), 1-25.

Knuth, D. E. (1996). Guest Lecture. *Preceding Graph Drawing 1996.*

Lam, S., & Sethi, R. (1979). Worst case analysis of two scheduling algorithms. *SIAM Journal on Computing, 6*(3), 518.

Lee, Y. Y., Lin, C. C., & Yen, H. C. (2006). Mental map preserving graph drawing using simulated annealing. *Proceedings of the Asia Pacific Symposium on Information Visualisation, Japan, 60,* 179-188.

Li, X. U., & Stallmann, M. (2002). New bounds on the barycenter heuristic for bipartite graph drawing. *Information Processing Letters, 82*(6), 293-298.

Lin, X. Y. (1992). *Analysis of algorithms for drawing graphs.* Unpublished doctoral dissertation, University of Queensland, Queensland, Australia.

Luder, P., Ernst, R., & Stille, S. (1995). An approach to automatic display layout using combinatorial optimization algorithms. *Software: Practice and Experience (25)*11, 1183-1202.

Marti, R., & Laguna, M. (2003). Heuristics and meta-heuristics for 2-layer straight line crossing minimization. *Discrete Applied Mathematics, 12*(3), 665-678.

Matuszewski, C., Schönfeld, R., & Molitor, P. (1999). Using sifting for k-layer straightline crossing minimization. *Proceedings of the 7th International Symposium on Graph Drawing, England, 1731,* 217-224.

Miriyala, K., & Tamassia, R. (1993). An incremental approach to aesthetic graph layout. *Proceedings of the Sixth International Workshop on Computer-Aided Software Engineering, Singapore, 47*(11), 1297-1309.

North, S. C. (1995). Incremental layout in DynaDAG. *Software and Systems Research Center.* AT & T Bell Laboratories.

North, S. C., & Woodhull, G. (2001). On-line hierarchical graph drawing. *Lecture Notes in Computer Science; Vol. 2265: Revised papers from the 9th international symposium on graph drawing* (pp. 232-246). London: Springer-Verlag.

Patarasuk, P. (2004). *Crossing reduction for layered hierarchical graph drawing.* Unpublished master's thesis, Florida State University, Tallahassee, Florida.

Rabani, Y. (2003). *Approximation Algorithms Lectures*. Retrieved May 2, 2007, from http://www.cs.technion.ac.il/~rabani/236521.04.wi.html

Raitner, M. (2004). *Maintaining hierarchical graph views for dynamic graphs* (Tech. Rep. No. MIP-0403). University of Passau, Germany.

Rudell, R. (1993). Dynamic variable ordering for ordered binary decision diagram. In *Proceedings of the International Conference on Computer-Aided Design* (pp. 42-47). Los Alamitos, CA: IEEE Computer Society Press.

Ryall, K., Marks, J., & Shieber, S. (1997). *An interactive constraint-based system for drawing graphs*. Mitsubishi Electric Research Laboratory.

Sander G. (1996). *Visualisierungstechniken für den Compilerbau*. Unpublished doctoral dissertation, University of Saarbrücken, Germany.

Stallman, M., Brglez, F., Ghost, D. (2001). Heuristics, experimental subjects, and treatment evaluation in bigraph crossing minimization. *Journal of Experimental Algorithmics, 6*(8).

Stedile, A. (2001). *JMFGraph - A modular framework for drawing graphs in Java*. Unpublished master's thesis, Graz University of Technology, Graz, Austria.

Sugiyama, K., Tagawa, S., & Toda, M. (1981). Methods for visual understanding of hierarchical systems. *IEEE Trans. on System, Man, and Cybernetics,* (2), 109-125.

Waddle, V. E. (2001). Graph layout for displaying data structures. In J. Marks (Ed.), *Proceedings of the 8th International Symposium on Graph Drawing* (pp. 241-252). London: Springer-Verlag.

Weisstein, E. W. (2003). Independent set. *MathWorld--A Wolfram Web Resource*. Retrieved October 20, 2006, from http://mathworld.wolfram.com/IndependentSet.html

West, D. B. (2001). *Introduction to graph theory* (2nd ed.). New Jersey: Prentice Hall.