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

by

Dung Mai

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

Graduate School of Computer and Information Sciences
Nova Southeastern University

2011

Dissertation Signature (Approval) Page

We hereby certify that this dissertation, submitted by Dung Mai, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.

Michael J. Laszlo, Ph.D. Chairperson of Dissertation Committee	Date
Wei Li, Ph.D. Dissertation Committee Member	Date
Junping Sun, Ph.D. Dissertation Committee Member	Date

Approved:

Amon B. Seagull, Ph.D. Interim Dean, Graduate School of Computer and Information Sciences	Date

Graduate School of Computer and Information Sciences
Nova Southeastern University
2011

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
September 2011

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 change rapidly in both content and quantity, 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 generated by 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 were developed in this research. The goal of this research was to improve the scalability of the constrained graph drawing framework while preserving layout stability. The framework's use of the relational data model shifts the graph application from the traditional desktop to a collaborative and distributed environment by reusing vertex and edge information stored in a relational database. This research was based on the work of North and Woodhull (2001) and the constrained crossing reduction problem proposed by Forster (2004). The result of the constrained hierarchical graph drawing framework and the new Sugiyama heuristic, especially the modified barycenter algorithms, were tested and evaluated against the Graphviz framework and North and Woodhull's (2001) online graph drawing framework.

The performance test results showed that the constrained graph drawing framework run time is comparable with the performance of the Graphviz framework in terms of generating static graph layouts, which is independent of database accesses. Decoupling graph visualization from the graph editing modules improved scalability, enabling the rendering of large graphs in real time. The visualization test also showed that the constrained framework satisfied the aesthetic criteria for constrained graph layouts. Future enhancements for this proposed framework include implementation of (1) the horizontal coordinate assignment algorithm, (2) drawing polylines for multilayer edges in the rendering module, and (3) displaying subgraphs for very large graph layouts.

Acknowledgments

The completion of this dissertation would not have been possible without the support and encouragement from many.

First and foremost I would like to express my deep gratitude and sincere appreciation to my advisor, Dr. Laszlo, whose patience, guidance, and support help me complete this dissertation. I also extend my appreciation to the dissertation committee members Dr. Sun and Dr. Li for their support and expertise in serving as part of the committee for my dissertation.

I must express special thanks to my wife, Ha Mai, who always believed I could complete this undertaking. This work could not have been completed without her endless patience, support, and encouragement. I also extend my utmost gratitude to my mother-in-law, Vietnga Pham, for staying with us in the past few years to help us take care of my daughters so I could work on my dissertation. I also extend my many appreciation to my former co-worker and a close friend, Mr. Webber, for not only his endless help in editing my dissertation but also for his great suggestions. I also would like to thank Mr. Evans for providing a foundation for the graph rendering engine. I thank my daughters for their understanding and curiosity about science that motivated me to finish this project.

I would like to also express thanks to all my brothers, my relatives, friends, and co-workers for their encouragement and support in this degree work.

I would like to dedicate this thesis to my father, who endured and overcame the hardship in the communist re-education camp, and to my mother who worked so hard to raise her four children while her husband was held indefinitely in the communist prison. Their lives have been and will always be a beacon that gives light, hope, encouragement, and motivation to my life.

Last but not least, I would also dedicate this thesis in the memory of my uncle, my mother's older brother, whom I wanted to see some time in the future but passed away abruptly while I was working on this thesis.

Table of Contents

List of Tables

List of Figures

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).

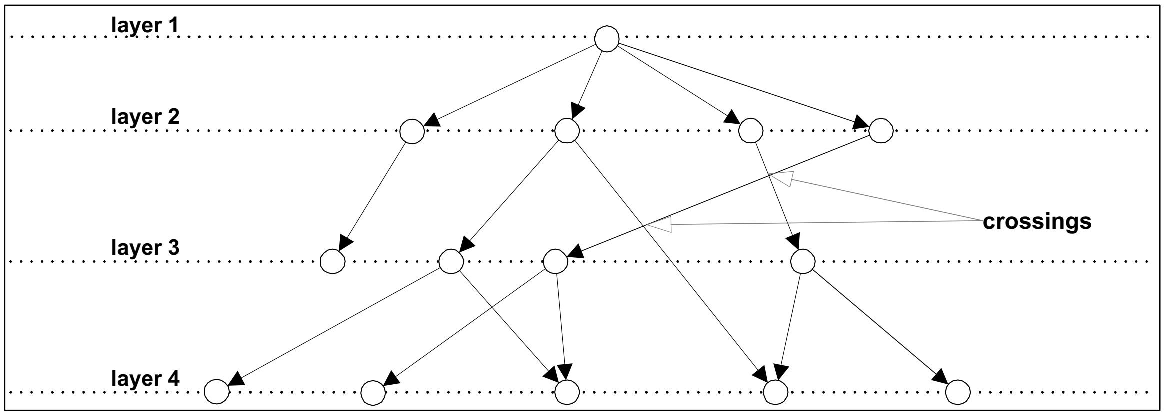


Figure 1. A hierarchical graph layout

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 disrupted if the next 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 sometimes can 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 on line. To support both editing and viewing in real-time mode, graph editing applications need to not only generate incrementally stable layouts but to 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) propose dynamic graph algorithms for different types of graph drawing techniques, especially series-parallel directed graphs and trees. Miriyala and Tamassia (1993) introduce an incremental graph drawing algorithm for drawing orthogonal graph layouts. Brandes and Wagner (1997) formulate dynamic layouts in terms of *random fields* and present a formal concept for dynamic graph layouts. He and Marriott (1998) propose several models for constrained force directed graph layout. Diehl, Görg, and Kerren (2000) present 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) present a modified simulated annealing algorithm that preserves the user's mental map and ensures graph layout stability. Frishman and Tal (2007) propose 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 discusses 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 addresses 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 introduces incremental graph layout algorithms as a solution to this problem. The next section discusses the impacts and potential complexity created by incremental graph drawing algorithms, especially crossing reduction for one-sided two-layered graphs.

Problem Statement

While incremental graph drawing algorithms such as online dynamic graph drawing (North & Woodhull, 2001) for drawing hierarchical graph layouts do improve layout stability, 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.

The extent to which the number of crossings is minimized is one of the important criteria 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 aesthetic criteria 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 thesis was to develop an incremental graph drawing framework based on online dynamic graph drawing (North, 1995; North & Woodhull, 2001). The result of this research improved the incremental stability and scalability of graph drawing systems by utilizing relational data model to capture incremental graph layouts.

Goal

The objective of this research was (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 result of the research extended the North and Woodhull (2001) research and developed a variant version of the online dynamic graph drawing system, achieving a solution that reduced the number of crossings while accommodating the set of constraining criteria proposed by North (1995). The specific goals of this research were as follows:

(1) To 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 pseudovertex and a given set of edges connecting the new node to existing vertices.
   b. Deleting a vertex and all its incident vertices.
   c. Adding an edge.
   d. Deleting an edge.
   e. Adding ordered constraint, which ensures layout stability by enforcing the order of certain pairs of vertices on the same layer.

f. Removing ordered constraints.

(2) To formulate a parameterized equation that accommodates aesthetic criteria. This equation factors in the number of crossings for the one-sided two-layered crossing reduction problem.

(3) To develop a relational data model to capture incremental graph layouts.

(4) To develop a heuristic to solve this crossing reduction problem.

(5) To 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 also getting 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. Bohringer and Newbery (1990) proposed using constraints that are defined by the user or are based on previous layouts to improve the stability of the layouts. Cohen et al. (1992) proposed a dynamic algorithm for drawing planar graphs for a variety of standard drawings and defined a property for dynamic graph layout called *smooth update. L*uder, Ernst, and Stille (1995) present 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 drawing 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 and Kerren (2000) introduced an off-line dynamic graph layout algorithm called *foresighted layout* that preserves layout stability based on 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 this dissertation is the development of 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 the dynamic criteria with the crossing reduction problem posed challenges for the thesis. There was an inherent trade-off between satisfying the layout stability aesthetic criterion and reducing the number of crossings or readability criterion. Preserving layout stability could have increased the number of crossings, but reducing crossings could have compromised the aesthetic criteria (North & Woodhull, 2001; Forster, 2004; Görg, 2005).

Another trade-off was between the aesthetic criteria and the space-time complexity. Satisfying the aesthetic criteria could have increased 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 sought a solution that balances layout stability with minimum edge crossing numbers.

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 related to the inconsistency between theoretical results and real-world applications. Results from several experiments showed that some proposed heuristics solving the problem had good performance when tested using synthetic data but had poor performance using real-world data (Marti & Laguna, 2003). For instance, the median approach had better worst-case scenario efficiency than the barycenter approach, but the barycenter method outperformed the median method in practice.

One issue related 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 was challenging. Results from several experiments also indicated that experimental results using synthetic graph data did not necessarily reflect those of real-world graph applications (Marti & Laguna, 2003). Thus, generating graph data closely similar to real-world graph applications posed 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) showed 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 posed an interesting problem.

Most researchers assumed that minimizing the number of edge crossings will improve the readability of the layout, but the recent study by Huang and Eades (2005) showed that in some cases graph layouts with more edge crossings due to some constraints are easier to understand than the same layout with fewer edge crossings. The result of their experiments showed 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 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 was 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 was 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 was used for testing the proposed system and the heuristic for the constrained one-sided two-layered crossing reduction problem either came from public domains or was 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 used in this thesis did not take into account possible concurrent execution of graph structure edits. This feature is discussed in this report as a further enhancement.

Though real-world graph data have different types of shapes and sizes, for this thesis the incremental hierarchical graph drawing framework assumed that the sizes were zero and shapes were simple circles. However, the framework was designed to be extensible and to accept different sizes and shapes of vertices. The enhancement of the framework is presented in the Recommendations section of this dissertation.

Definitions of Terms

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

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

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

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

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 are 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 where the positions are numbered from left to right. Formally, the barycenter value of a vertex u on layer L_1 can be defined as follows:

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

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

Complete graph: A *complete graph* is a simple graph such that each pair of vertices is connected by an edge.

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:

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

Curve: A curve δ is a continuous mapping to topological space *S* such that $\delta : I \rightarrow S$, 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 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 acyclic graph (DAG): A directed graph that has no cycles.

Directed graph with cycles: Let *G (V, A)* be a directed graph. *G* has a cycle if $\forall R \subset A \mid R$ 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: $D = \frac{2|E|}{|V|(|V|-1)}$, 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 $A \cap B = \emptyset$, 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 $a \leq b \leq c \leq d$.

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

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 2 shows a *two-partite*, or *bipartite*, graph with two independent sets of vertices: *(a, b, c)* and *(d, e, f, g, h)*.

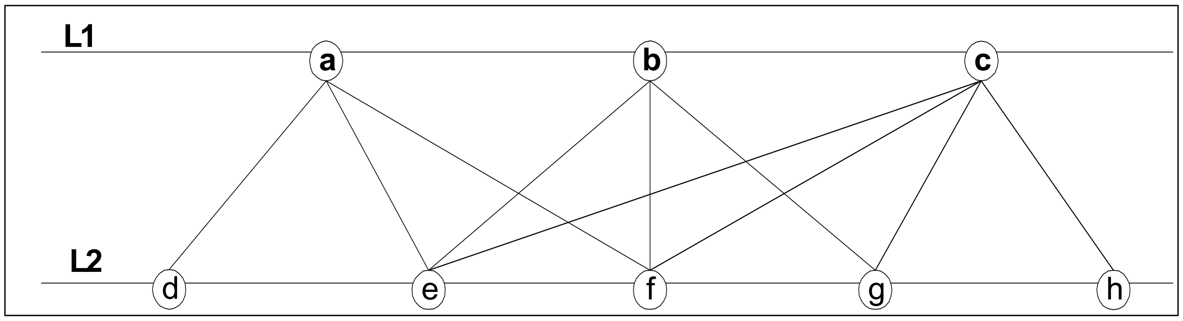


Figure 2. A bipartite graph

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 $L_1, L_2, L_3, \ldots L_k$ 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 3 shows a drawing of a four-layered hierarchical graph that has a height of *4* and a width of *5*.

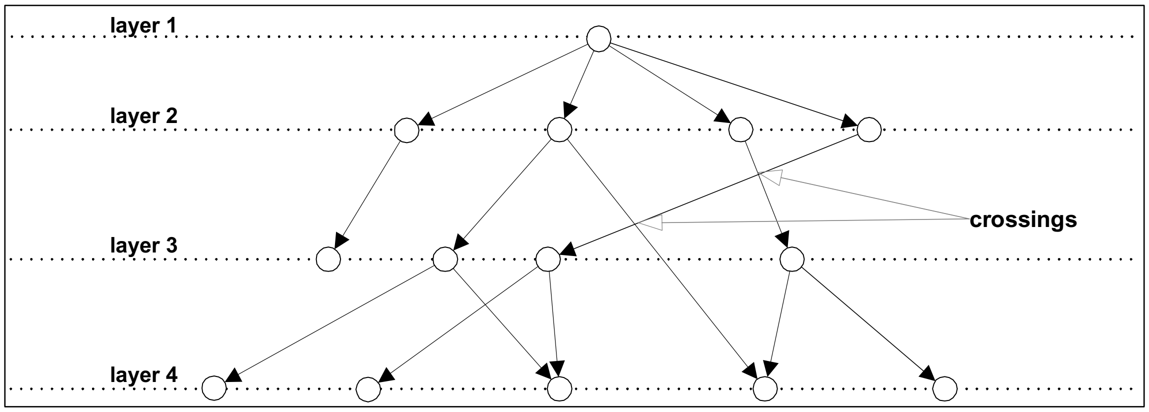


Figure 3. A 4-layered graph layout (Battista et al., 1999)

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:

$(a,b) \leq (a',b')$ iff $a < a'$ *or* $a = a'$ and $b \leq b'$

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 $u'_1, u'_2, \ldots, u'_n$ on layer L_2, with $pos(u'_1) < pos(u'_2) < \ldots < pos(u'_n)$, 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 that are adjacent to *v*, written as *N(v)* (West, 2001).

Off-line dynamic graph layout: Given a sequence of n graphs $g_1, g_2, \ldots, g_n$. Compute layouts $l_1, l_2, \ldots, l_n$ for these graphs such that

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

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

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 $g_1, g_2, \ldots, g_n$. Compute layouts $l_1, l_2, \ldots, l_n$ 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*, $V_i = \{v_1, \ldots, v_{ni}\}$ are vertices of layer *i*. $pos: V_i \rightarrow (1, \ldots, n_i)$ 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 a 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.

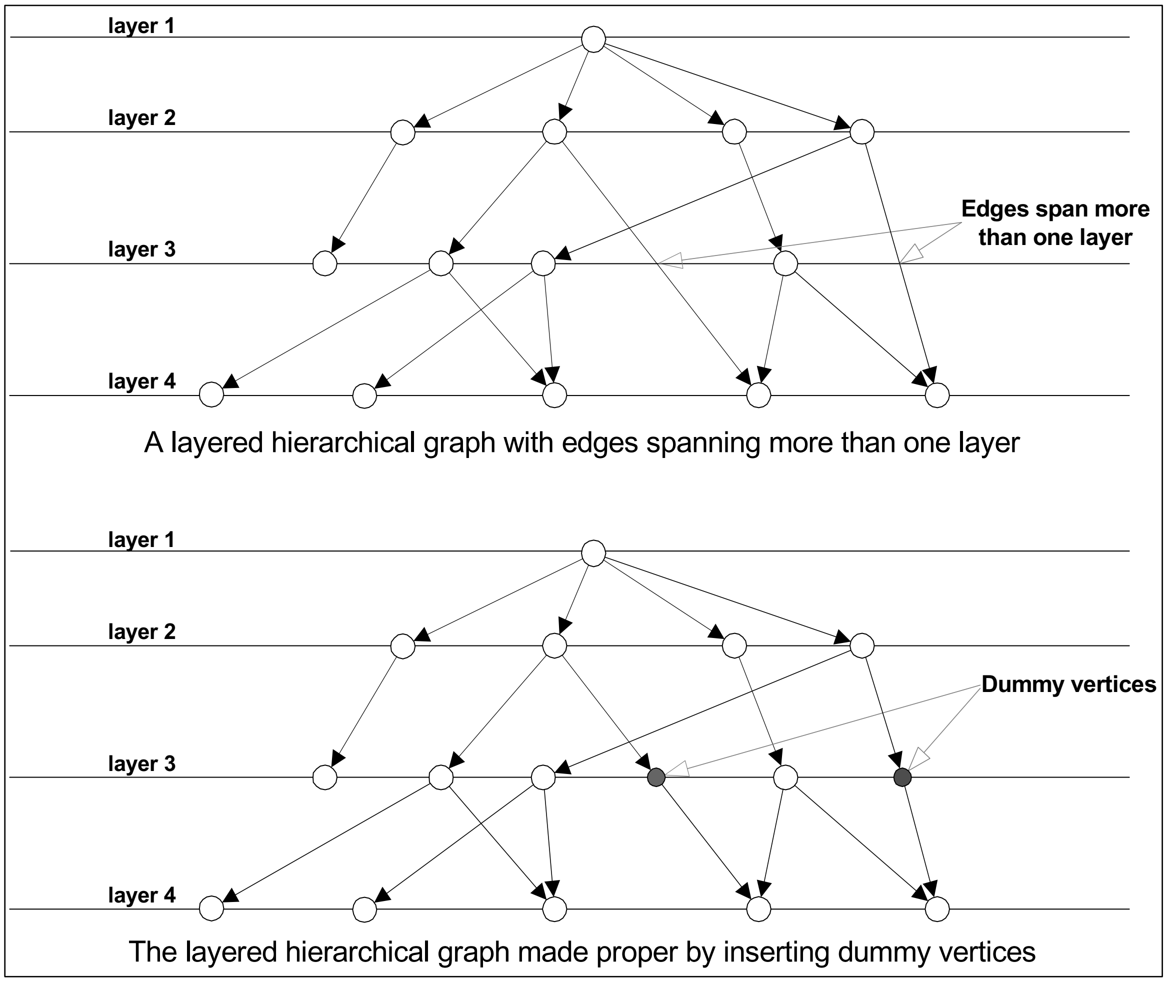

layer 1
layer 2
layer 3
layer 4
Edges span more than one layer
A layered hierarchical graph with edges spanning more than one layer
layer 1
layer 2
layer 3
layer 4
Dummy vertices
The layered hierarchical graph made proper by inserting dummy vertices

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 quality of a layout is the number of crossings; the fewer the crossings, the higher the quality.

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

(1) $\overline{\Delta} = \sum_{1 \le i < n} \Delta(l_{i-1}, l_i)$ 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 layout aesthetics, property (1) is equivalent to preserving the mental map of the layout and (2) is equivalent to reducing the number of edge crossings in the layout. These two goals often contradict one another.

Ratio bound performance of one-sided crossing reduction heuristics: Let G = *(L₁, L₂, E)* be a two-layered hierarchical graph, $u, v \in L_1$, pos_1, pos_2 be the ordering of layers L_1 and L_2 respectively, and pos_2 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 $h = \dfrac{\mathrm{opt_h}\,(\mathrm{G}, \mathrm{pos_2})}{\mathrm{LB}\,(\mathrm{G}, \mathrm{pos_2})}$ *(1)*

In which $\mathrm{LB}\,(\mathrm{G}, \mathrm{pos2}) = \sum_{u,v \in L_1} \min(c_{uv}, c_{vu})$ (2) as defined in the *Trivial lower bound of one-sided two-layered graph crossing reduction problem* definition.

(1) & (2) $=>$ Ratio bound of $h = \dfrac{\mathrm{opt_h}\,(\mathrm{G}, \mathrm{pos_2})}{\sum_{u,v \in L_1} \min(c_{uv}, c_{vu})}$

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.

R-approximation algorithm: A polynomial-time algorithm that produces a solution at most *r* times the optimum for a minimization problem (Rabani, 2003).

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

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

Source: A vertex that has outgoing edges but has 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 layout.

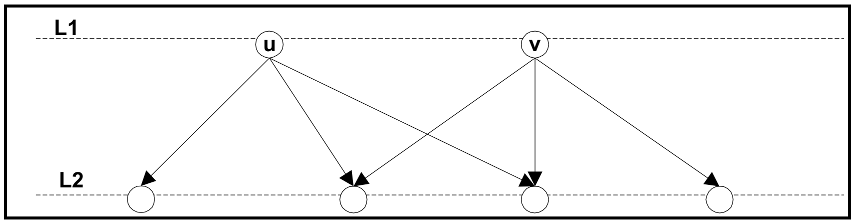


Figure 5. A two-layered hierarchical graph layout

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 $opt(G, pos_2)$ 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.

*F*ind an ordering pos_1 such that $crossing\ (G, pos_1, pos_2) = opt(G, pos_2)$.

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, $u, v \in L_1 \,/\, pos(u) < pos(v)$, C_{uv} is defined as the crossing number of edges incident with *u* and edges incident with *v*.

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

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 the positions of *u* and *v*, where those positions are numbered from left to right, 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.

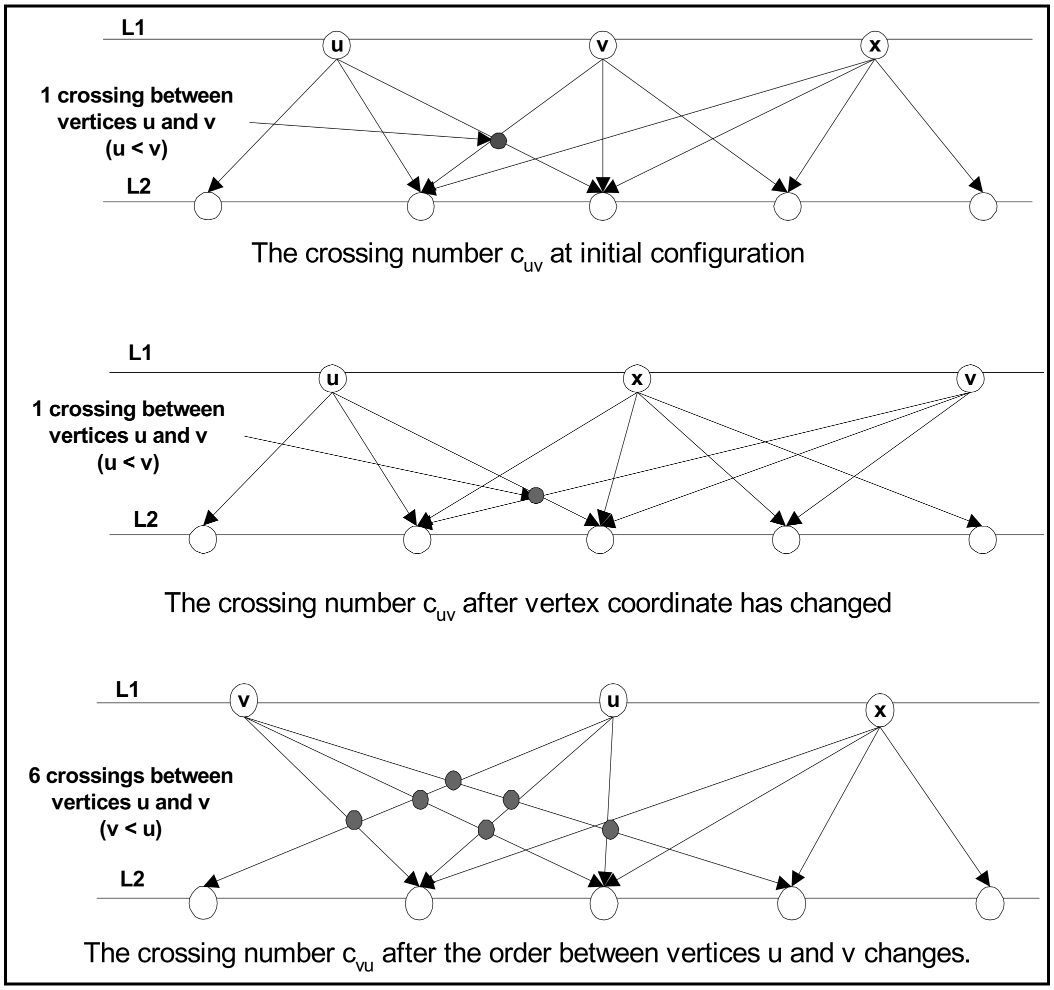

L1
u
v
x
1 crossing between vertices u and v (u < v)
L2
The crossing number c_{uv} at initial configuration
L1
u
x
v
1 crossing between vertices u and v (u < v)
L2
The crossing number c_{uv} after vertex coordinate has changed
L1
v
u
x
6 crossings between vertices u and v (v < u)
L2
The crossing number c_{vu} after the order between vertices u and v changes.

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* $(D(G), pos_1, pos_2) = \sum_{u,v \in L_1 | pos(u) < pos(v)} c_{uv}$ (1)

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

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

The remainder of this report 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 4 presents the test results. Chapter 5 provides the conclusion of this research.

Chapter 2

Review of the Literature

Introduction

The work of this research was 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 is 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 steps, 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. At the end of each section and subsection is a table that summarizes the characteristics of the different algorithms.

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 step. 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. The optimal solution for the graph in Figure 7 is to reverse set {(9, 4), (11, 5)}. Figure 8 shows that the directed graph is acyclic after reversing the directions of edges (9, 4) and (11, 5).

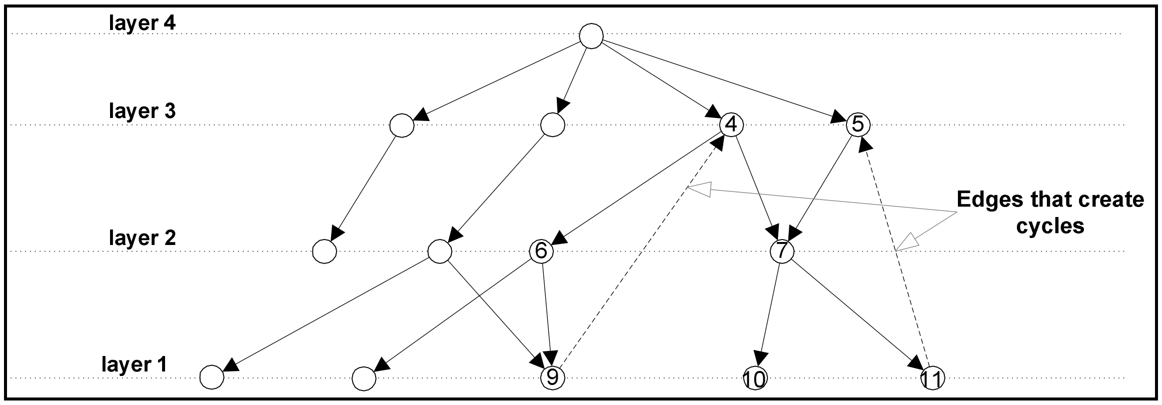


Figure 7. A directed graph with cycles

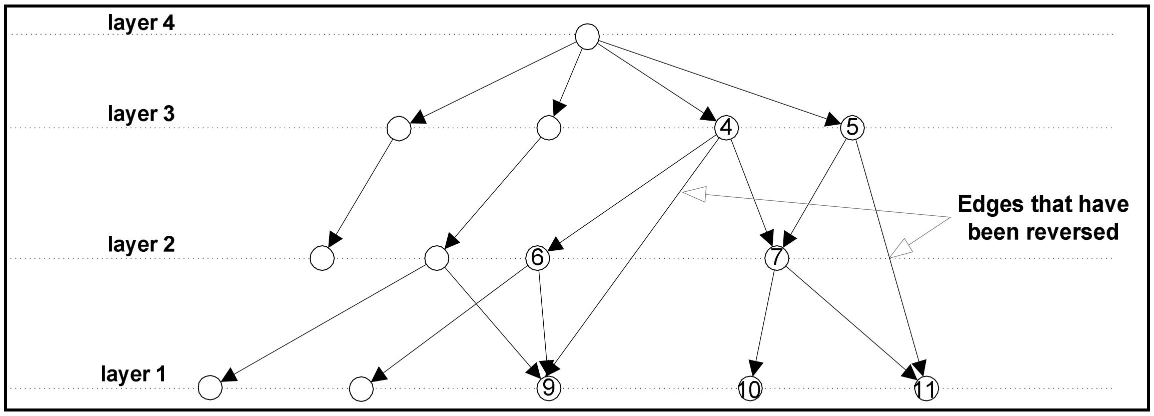


Figure 8. An acyclic directed graph after reversing the set of edges {(9, 4), (11, 5)}.

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), and 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 the Approximation algorithm (Berger & Shor, 1990), which could provide an optimal solution but 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) presented an approximation algorithm based on the *Local-Ratio* technique, which provided an approximation algorithm for the *covering* problem. The approximation algorithm consisted of two phases. The first phase searched for simple cycles C in the directed graph. If such a cycle existed, the algorithm identified edges in C whose weight, denoted as ε, was minimal. Then the weight of all the edges in C was reduced by ε and the edges with a weight of zero were removed. If no more cycles were found the first phase terminated. The second phase was to add some deleted edges back to the graph without re-creating cycles. The approximation ratio of the algorithm was bounded by the length of the longest simple cycle of the directed graph. However, the proposed algorithm worst-case run time was $O\ (|V| \times |E|)$.

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	O (|V| + |E|)	Result may be poor
Greedy-Cycle-Removal	Greedy	O (|V| + |E|)	Result is good and the run time is linear
Approximation algorithm	Local search	O (|V| × |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 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 multiprocessor scheduling. The Coffman-Graham layering algorithm (Coffman & Graham, 1972) for solving multiprocessor 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 the worst-case scenario the height will not exceed twice the optimal height when $w \to \infty$, as indicated in the following equation: $h \leq (2 - \frac{2}{w}) \times h_{min}$, 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 used in this dissertation employed 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 number of 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, reordering 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 9 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.

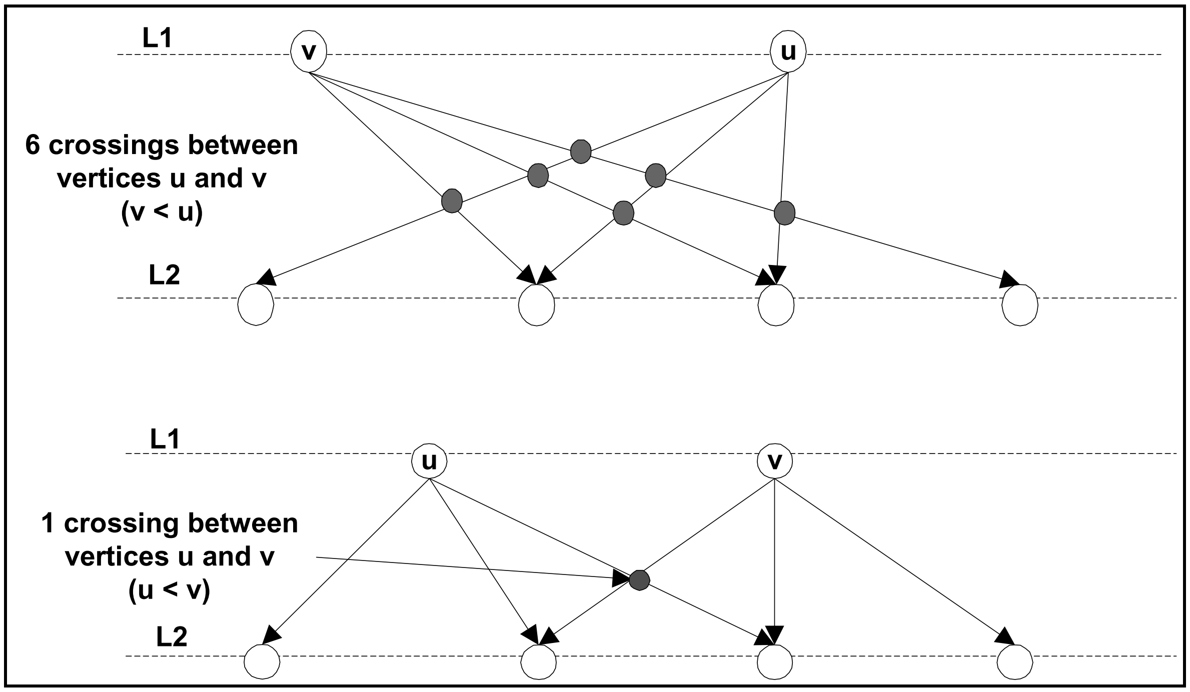

L1
v
u
6 crossings between vertices u and v (v < u)
L2
L1
u
v
1 crossing between vertices u and v (u < v)
L2

Figure 9. A two-layered hierarchical graph layout 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 the 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 the ratio of the number of edge crossings produced by the algorithm and the minimum number of edge crossings of the barycenter heuristic is $(\sqrt{|V|})$ (Li & Stallmann, 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 the 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 the 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., 2002). 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: $runtime = \frac{CN}{LB}$, 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 its left neighbors. 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 focused 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 the layout stability. Reducing edge crossings for one-sided two-layered graph layouts with vertex constraints is called *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 $C \subseteq L_1 x L_1$. 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 pos(u) < pos(v). The constraint c(u,v) is satisfied when pos(u) < pos(v) and is violated when the pos(u) > pos(v).

Sander (1996) proposed a simple solution for solving crossing reduction for constrained one-sided two-layered graph layout. 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 either 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 similar solution to that of Sander (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 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 layout to a *weighted feedback arc se*t problem. The heuristic first constructs a *penalty graph*, which is a mapping of 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 than 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 vertex *v* indicates more edges are to the left of the vertex than to its right. Forster (2004) proposed 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 one-sided two-layered crossing reduction problem.

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	$O(\|V\|\log\|V\|)$	Good performance comparing to barycenter and median
Greedy-switch (Eades & Kelly, 1986)	Scan vertices and compare the crossing numbers	$O(\|V\|\log\|V\|^2)$	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	$\rho = \frac{CN}{LB}$	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	$O(\|V\|^2)$	Outperformed by barycenter heuristic
Adaptive insertion (Stallman et al., 2001)	Local search	$O(\|V\| \times \|E\|)$	Not scalable for large graphs
Penalty graph (Demetrescu & Finocchi, 2003)	Induce as a feedback arc set problem	$O(\|V\|^4 + \|E\|^2)$	Provides better drawing with fewer crossings but is not scalable for large graphs
Modified Barycenter (Forster, 2004)	Sorting vertices	$O(\|V\| \log \|V\| + \|E\| + \|C\|^2)$	Provides layout as good as those of other complicated algorithms but with a better run time
Modified Barycenter Sander (1996)	Sorting vertices	Near linear	Results are sometimes not as good as the layout without constraints
Modified Barycenter Waddle (2001)	Sorting vertices	Near linear	Results are worse than that 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 proposed layout 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. Th*is 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) that 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 combinatoric 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 drawing framework, where three models are for undirected graphs and one is for trees.

Diehl and Kerren (2000) pointed out the disadvantages of graph animation and online dynamic graph layout. Graph animation technique simply shows that 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 drawing framework.

Table 4. Incremental graph drawing frameworks.

Name	Approach	Graph Type	Note
Bohringer and Newbery (1990)	Online dynamic graph layout	Generic graphs	Address layout stability but the framework is not scalable
Cohen et al. (1992)	Online dynamic graph layout	Tree, series-parallel digraphs	
Luder et al. (1995)	Online dynamic graph layout	Generic	Provide interactive graph drawing environment. However, it can handle a graph with only up to 50 vertices.
Ryall et al. (1997)	Online dynamic graph layout	Generic	Interactive diagram editor for drawing small graphs
Brandes and Wagner (1997)	Online dynamic graph layout	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	Preserve 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	Preserve the graph model by using a data structure to capture the graph attributes. Efficient and scalable but does not address the constrained crossing reduction problem.
Lee et al. (2006)	Online	Simulated annealing	Allow users to adjust the relative weight of aesthetic criteria
Frishman and Tal (2007)	Online	Directed force	Very scalable and effective for directed force graph layouts

Summary

This section reviewed 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 showed 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.

Contribution to the Field

The work of this dissertation made 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 using a relational database, and (2) developing a method to solve the constrained crossing reduction problem for dynamic hierarchical graph layouts based on the work of Forster (2004) .

Chapter 3
Methodology

Introduction

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

1. The first task was to develop a formal model for incremental graph layout. The objective of this model was to balance layout stability with readability criteria. This task was based on the works of North and Woodhull (2001) and Görg (2005).
2. The second task was to design and develop a constrained hierarchical directed graph drawing and visualization framework that extended and enhanced the online graph drawing framework proposed by North and Woodhull (2001). The developed framework included several new functions. The first function was to preserve the states of the incremental graph layouts in a relational database. This enabled the 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

decoupled the visualization component from the editing component. This separation enabled 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 framework enabled end users to input constraints to the layouts. These user constraints influenced how the graph layout was generated based on the user's input.

3. The third task was 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 modified crossing reduction algorithm incorporated the user's constraints to ensure graph layout stability. The goal of the modified crossing reduction algorithm was to find the optimum balance between layout stability and readability. The algorithm was based on the work of Forster (2004).
4. The fourth task was to evaluate the performance and efficiency of the new algorithm. This involved 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

The chapter first reviews the aesthetic criteria for hierarchical graph layouts. Second, it reviews the research that contributed to the work of this thesis, as shown in Figure 10. Reviewing North and Woodhull's (2001) online graph drawing framework and aesthetic criteria provided 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. Finally, the chapter describes experimental procedures and ends with a chapter summary.

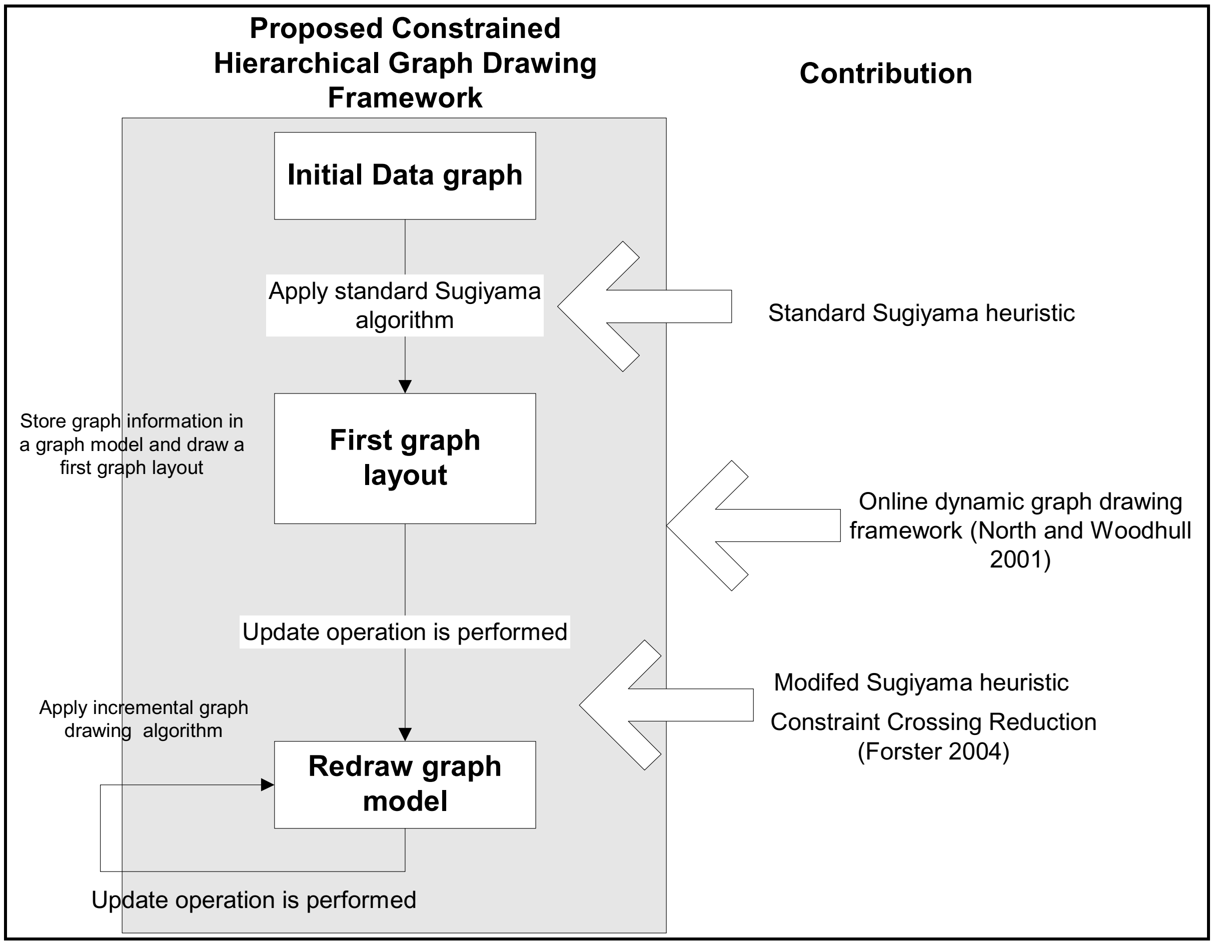

Proposed Constrained Hierarchical Graph Drawing Framework
Contribution
Initial Data graph
Apply standard Sugiyama algorithm
Standard Sugiyama heuristic
Store graph information in a graph model and draw a first graph layout
First graph layout
Online dynamic graph drawing framework (North and Woodhull 2001)
Update operation is performed
Modifed Sugiyama heuristic
Constraint Crossing Reduction (Forster 2004)
Apply incremental graph drawing algorithm
Redraw graph model
Update operation is performed

Figure 10. Proposed constrained hierarchical graph drawing system and contributing research

Assumptions and Standard Notations

For conciseness, the proposed constrained incremental graph drawing and visualization framework is denoted as the *constrained graph drawing framework* and is 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) also 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. note, 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 $G_1, G_2, \dots\dots\dots, G_n$. *The* generated drawings of these successive versions of *G* is also a series of drawings $L_1, L_2, \dots\dots\dots, L_n$ (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 the 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, as illustrated in Figure 11. 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.

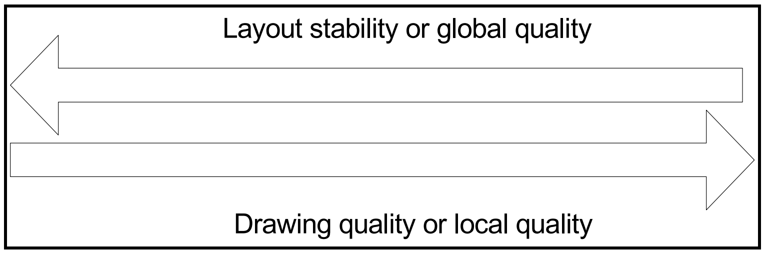

Layout stability or global quality
Drawing quality or local quality

Figure 11. Two conflicting goals of incremental graph layout (Görg, 2005)

The aesthetic criteria described in this section influenced the design of the constrained incremental graph drawing framework by being utilized in developing the formal layout model. In the next section we review the research on the constrained graph drawing framework, as this review helped 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

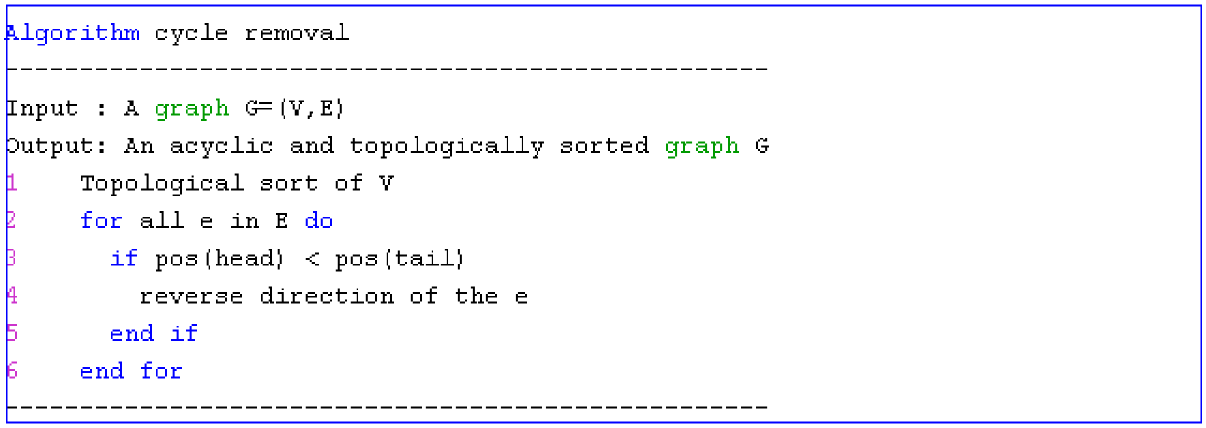
The Standard Sugiyama Heuristic

As our constrained graph drawing framework was designed to utilize algorithms in the Sugiyama heuristic in building initial graphs and updating the graph layout due to dynamic operations, this section discusses in detail each algorithm that was used in our constrained graph drawing framework and presents the pseudocode of those algorithms.

Step 1: Cycle Removal

Chapter 2 introduced the brute force algorithm, DFS (Depth First Search) or BFS (Breath First Search) algorithms, the penalty graph (Finocchi, 2002), and the Greedy-Cycle-Removal algorithm (Eades et al., 1993). Among these algorithms, DFS/BFS and Greedy-Cycle-Removal algorithms have linear run time. Although in theory the DFS/BFS could produce poor results, our preliminary tests showed that a variation of the DFS/BFS algorithms produced a good result for graph layouts that have no sources and sinks but have strong connected edges. On the other hand, our implementation of the Greedy-Cycle-Removal algorithm produced a sorted list that was different from a sorted list that would have been created based on the natural ordering that comes from the input. The difference in terms of sorting was due to the way the Greedy-Cycle-Removal algorithm chooses vertices. As a result, the constrained graph drawing framework implemented both DFS and Greedy-Cycle-Removal algorithms for reversing any cycles temporarily, as there was no clear indication which algorithm is a better choice for reserving the cycles. Furthermore, the constrained graph drawing framework was also designed to enable users to switch algorithms based on the generated graph layout. This section describes Greedy-Cycle-Removal and the modified DFS algorithms.

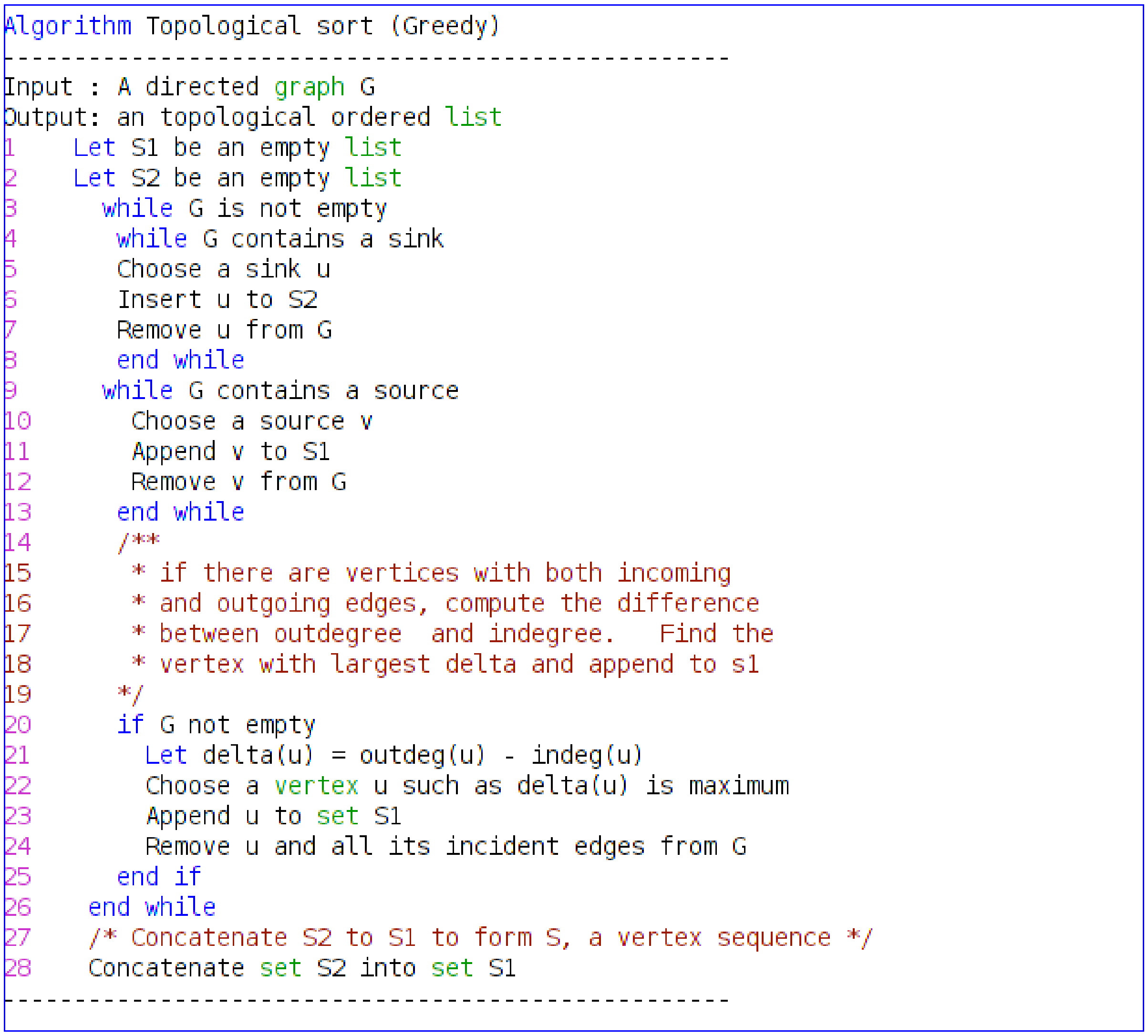
Since both DFS/BFS and Greedy-Cycle-Removal algorithms have the same goal--to reverse edges that produce cycles--the only difference between the DFS/BFS and Greedy-Cycle-Removal algorithms is how they sort the data. For readability purposes, the cycle removal algorithm is shown as a two-step procedure. The main procedure is to reverse the edges with cycles and the second procedure is a topological sort algorithm, which can be either a Greedy-Cycle-Removal or a DFS/BFS algorithm. The pseudocode of the main procedure is described as follows: First sort the list of vertices based on their topological values. Then reverse any edges whose sink's position is greater than the source's position in the sorted list. The pseudocode of the main cycle removal procedure is shown in Figure 12.



```
Algorithm cycle removal
----------------------------------------------------
Input : A graph G=(V,E)
Output: An acyclic and topologically sorted graph G
    Topological sort of V
    for all e in E do
      if pos(head) < pos(tail)
        reverse direction of the e
      end if
    end for
----------------------------------------------------
```

Figure 12. Pseudocode of the cycle removal algorithm

Greedy-Cycle-Removal is a 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 is described as follows: First create two empty lists, namely S_1 and S_2. While the graph is not empty, *append* sources to S_1 (add to the end of the list) and *insert* sinks into S_2 (insert at the beginning of the list). If there are more vertices, calculate the delta between the outdegree (number of outgoing edges) and indegree (number of incoming edges) of the remaining vertices. Append the vertex with the largest delta value to S_1. Finally, concatenate S_2 to S_1 to create a sequence of vertices. Figure 13 shows the pseudocode of the Greedy topological sort algorithm.



```
Algorithm Topological sort (Greedy)
---------------------------------------------------
Input : A directed graph G
Output: an topological ordered list
     Let S1 be an empty list
     Let S2 be an empty list
       while G is not empty
        while G contains a sink
        Choose a sink u
        Insert u to S2
        Remove u from G
        end while
       while G contains a source
         Choose a source v
         Append v to S1
         Remove v from G
        end while
        /**
         * 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
          Let delta(u) = outdeg(u) - indeg(u)
          Choose a vertex u such as delta(u) is maximum
          Append u to set S1
          Remove u and all its incident edges from G
        end if
      end while
      /* Concatenate S2 to S1 to form S, a vertex sequence */
      Concatenate set S2 into set S1
---------------------------------------------------
```

Figure 13. Pseudocode of the Greedy-Cycle-Removal algorithm (Eades et al., 1993)

The DFS algorithm is also used for sorting, but it randomly selects a starting point for the search without taking into account the source and sink. In a worst-case scenario, the DFS algorithm could produce a very poor result by reversing (m-1) edges where m is the number of edges. To avoid this potential pitfall, a variation of the DFS algorithm was implemented in the constrained graph drawing framework. The pseudocode of the modified DFS algorithm is described as follows: First create two empty lists, namely S_1 and *result*. While the graph is not empty, append sources to S_1. For each v in the S_1 list, recursively append v and all its children into the result list. If there are still vertices, add those isolated vertices into the result list. Figure 14 shows the pseudocode of the modified DFS (Depth First Search) algorithm.

```
Algorithm Topological sort (DFS)
---------------------------------------------------
Input : A directed graph G
Output: an topological ordered list
     Let S1 be an empty list
     Let result be an empty list
       while G is not empty
       while G contains a source
        Choose a source v
        Append v to S1
        Remove v from G
       end while
     end while

      for each v in S1
        recursively add v into result
      end for
      if G not empty
         add v into result list
      end if
---------------------------------------------------
```

Figure 14. Pseudocode of the modified DFS algorithm (Eades et al., 1993)

Step 2: Layer Assignment

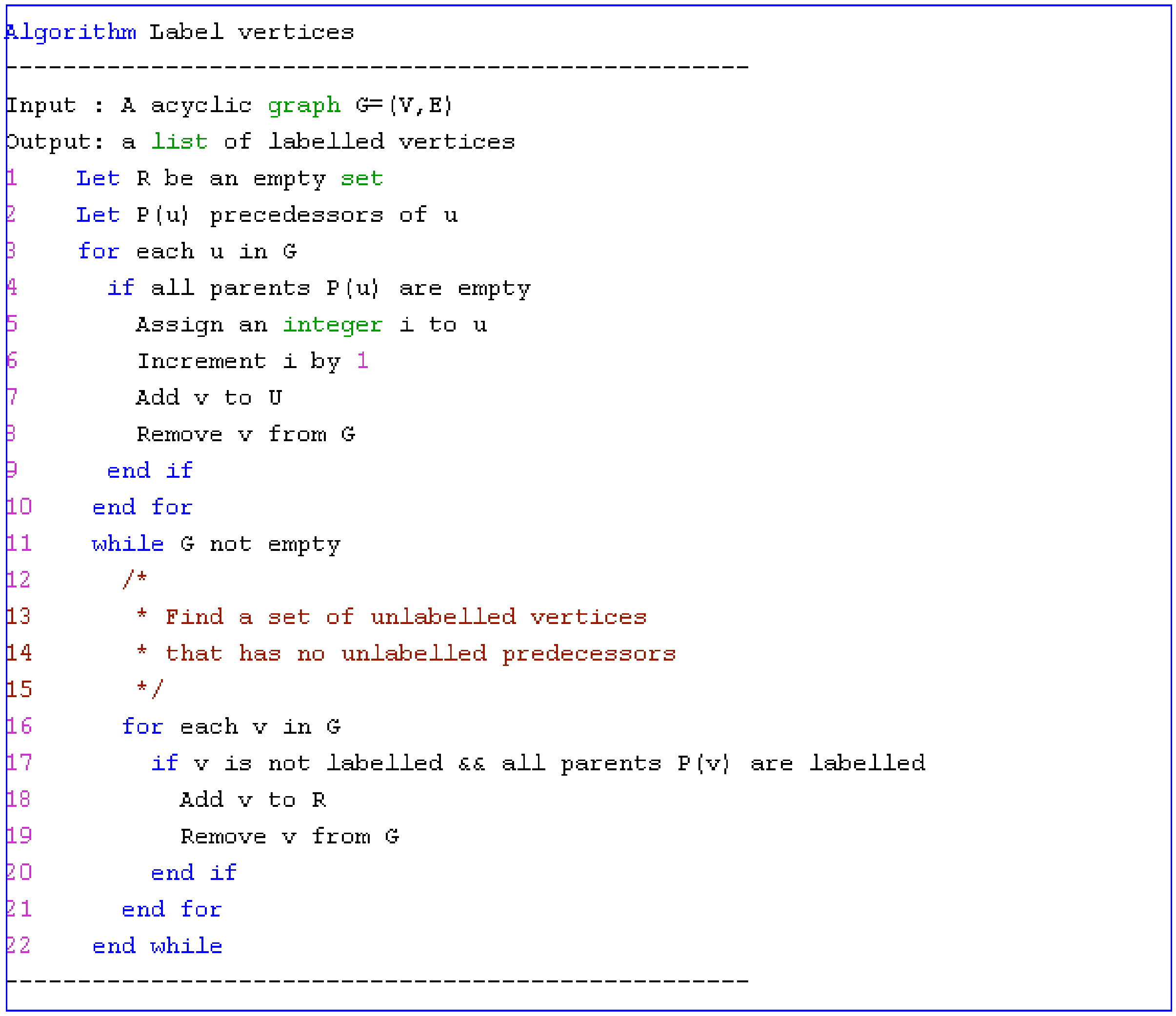
As discussed in Chapter 2, although the Longest Path Layering algorithm is simple and has a linear run time, this algorithm could produce a very wide layer. On the other hand, the Coffman-Graham layering algorithm (Coffman & Graham, 1972) also runs in linear but limits layout width. According to Battista 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 graph drawing framework was designed to 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 unlabeled. First it randomly selects a source and assigns an integer label *1* to that vertex. Then it loops through the remaining sources and assigns integer label $2,3, \ldots k$ to each source. For the remaining vertices in G, it performs the following procedure for assigning integer labels to vertices. This procedure first 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 and to assign vertices to layers as follows: First the algorithm sorts vertices based on their integer labels, and then it assigns vertices to layers, ensuring no layer receives more than W vertices where W is a constant 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 ($1 \leq \mathrm{i} < \mathrm{k}$). If the layer L_i width 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 assigned to a layer yet and all of its successors ($S(u)$) have been assigned to one of the layers $L_1, L_2, \ldots L_{k-1}$. If there is more than one such vertex, the procedure selects the vertex with the largest label. If there is no such vertex, or the width of layer L_k is larger than W, then it proceeds to the next layer L_{k+1}. This step is performed until all vertices are assign to appropriate layers. Figure 15 shows the pseudocode of the main Coffman-Graham algorithm, and Figures 16 and 17 show Label Vertices and Find Unassigned Vertices subroutines respectively.

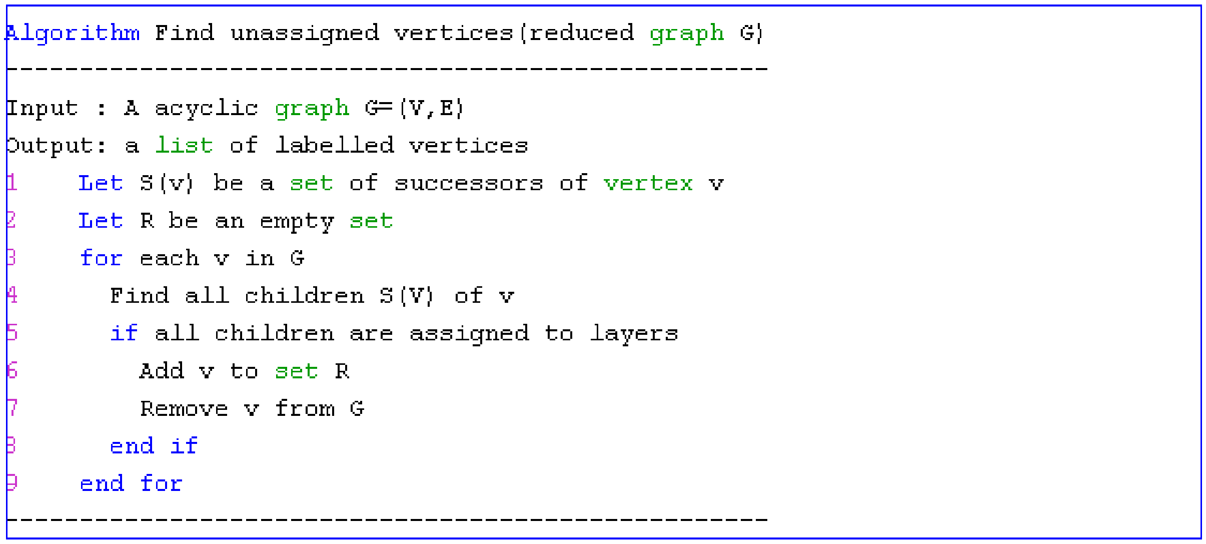
```
Algorithm Assign layer
----------------------------------------------------------
Input : Digraph G(V,E), positive integer W
Output: Proper layered digraph G(V,E)
    Let R(v) be a set of v
    Let i be a positive integer
    Let U be an empty set
    Let Li be layer i
    /* Step 1: assign integer labels to vertices. */
    Label vertices to have a set of labelled vertices U
     /* Step 2: Assign labelled vertices into layers */
     Assign sinks to layers (G, U) to  have a set of vertices
    while U not empty
       /**
         Find a set R of vertices that are not assigned
         to a layer yet
       */
       Find unassigned vertices (G, U) and put them into a set R
       Sort set R based on the vertex labels
         if R is empty then
           Add new layer
         else
           for each v in R
             Add v to Li
             if size of current layer >= W then
               Add new layer
             end if
           end for
         end if
     end while
----------------------------------------------------------
```

Figure 15. Pseudocode of the Coffman-Graham algorithm (Battista et al., 1999)



```
Algorithm Label vertices
-----------------------------------------------------
Input : A acyclic graph G=(V,E)
Output: a list of labelled vertices
    Let R be an empty set
    Let P(u) precedessors of u
    for each u in G
      if all parents P(u) are empty
        Assign an integer i to u
        Increment i by 1
        Add v to U
        Remove v from G
      end if
    end for
    while G not empty
      /*
       * Find a set of unlabelled vertices
       * that has no unlabelled predecessors
       */
      for each v in G
        if v is not labelled && all parents P(v) are labelled
          Add v to R
          Remove v from G
        end if
      end for
    end while
-----------------------------------------------------
```

Figure 16. Pseudocode of the Label Vertices algorithm (Battista et al., 1999)



```
Algorithm Find unassigned vertices(reduced graph G)
--------------------------------------------------
Input : A acyclic graph G=(V,E)
Output: a list of labelled vertices
    Let S(v) be a set of successors of vertex v
    Let R be an empty set
    for each v in G
      Find all children S(V) of v
      if all children are assigned to layers
        Add v to set R
        Remove v from G
      end if
    end for
--------------------------------------------------
```

Figure 17. Pseudocode of the Find Unassigned Vertices algorithm (Battista et al., 1999)

Step 3: Crossing Reduction

This step 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 crossings number is computed and is added to the total number of edge crossings. When it reaches the bottom, the algorithm moves upward and again computes the crossings number at each layer and adds the crossings number to the total number of edge crossings. Once reaching the top of the graph, the algorithm compares the previous total edge crossings number with the current total edge crossings 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 that the total edge crossings are fewer than the previous runs. Otherwise, the algorithm exits. The recent experiment done by Patarasuk (2004) showed that the numbers of crossings sometimes increases after a sweep but then decreases again after another sweep. Thus, there is no clear halt criterion in the layer-by-layer crossing reduction algorithm. 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 displayed in Figure 18.

```
Algorithm Layer to layer sweep
-----------------------------------------------------------
Input : Proper digraph G(V,E), maximum iteration M
Output: Proper digraph G(V,E) with fewer number of crossings
    Let prev_crossings be previous computed crossing numbers
    Let iter be the number of iterations
    Let C be the total number of edge crossings
    Let c be the number of edge crossings of two layers
    Let i be the integer number
    Let Li be the layer i
    while iter < M
      for each Li in G
      /* Compute the number of crossings for 2 layered graph */
       c = crossing number of (Li)
           C = C + c;
       end for
     /*
      * Compare against the previous calculation.
       * If the number of crossings is greater than previous number of
       * crossings, the procedure is done.
      */
       if C > prev_crossings
          exit the while loop
       /* If not, continue to calculate the crossing number */
       else
           prev_crossings = C
       end if
         Increment iter by 1
     end while
-----------------------------------------------------------
```

Figure 18. Pseudocode of the layer-by-layer sweep algorithm

To compute the edge crossings 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 was 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 Figure 19, and Figure 20 shows the subroutines that calculate the barycentric of vertices.

```
Algorithm Compute Crossing Number of two layered graph (layer Li, layer L(i+1))
---------------------------------------------------
Input : layer Li, L(i+1)
Output: number edge crossings
    Let crossings be a positive integer
    Let S be a set of vertices
    Compute bary center values for layer Li
    Sort set Li based on barycentric value of vertices
    for each vertex v in Li
      Set S = Li - v
      for each vertex v1 in S
        /**
        compute number of edge crossings between v and v1
         */
         crossings =
           crossings + edge crossings between (v and v1)
       end for
     end for
---------------------------------------------------
```

Figure 19. Pseudocode of the barycenter algorithm

```
Algorithm compute Barycenter (layer Li)
---------------------------------------------------
Input : layer Li
Output: list of vertices with bary centric value
    Let b(u) be barycentric value of a vertex u in  layer Li
    Let v be vertices on layer Li+1;
    Let Nu be neighbors of u in layer Li+1
    Let pos(v) be position of v on layer Li+1;
    for each u in Li
      /* Calculate barycentric value */
      b(u) = (Sum of pos(v)) / Nu
      end for
---------------------------------------------------
```

Figure 20. Pseudocode of the computing barycenter algorithm

Step 4: Coordinate Assignment

As mentioned in the limitations section of this report, the constrained graph drawing framework (CGDF) did not take into account the actual sizes and shapes of real-world vertices. All vertices were 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.) 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: (1) add a vertex, (2) add an edge, (3) remove a vertex and all its incident edges, (4) 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 layout and sends the result back to the client. The client will render the layout to reflect new changes. DynaDAG employs 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 and Woodhull 2001). The list of attributes is shown in Table 5. The proposed constrained graph drawing framework (*CGDF*) will employs a client-server model similar to that of DynaDAG but employs 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 *entity relationship* section.

Table 5. Internal Model used in the DynaDAG (North & Woodhull, 2001).

Value	Type	Explanation
G = (V;E)	graph object	graph
u, v, w… $\in$ V	vertex object	vertex
e,f, ….$\in$ E	Edge object	Edge
$\Delta(G)$	Coord	Minimum vertex separation
$L_{i,j}$	vertex object	jth node in ith 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 request
$X'(v), Y'(v)$	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 the DynaDAG is called *Process*, which has four phases similar to that of the Sugiyama heuristic. Each phase in the *Process* procedure examines subgraphs that are affected by *update* operations and 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 6. The constrained graph drawing framework (*CGDF*) used in this study supported similar operations, such as *insert*, *update*, and *delete* operations on subgraphs. However, implementation of the *CGDF* was different from that of the DynaDAG. While DynaDAG transforms Sugiyama phases into optimization problems and uses network simplex solver for solving those optimization problems, this study's *CGDF* employed modified algorithms that were widely used in each phase of the Sugiyama heuristic for solving these optimization problems.

Table 6. Objectives and constraints of the *Process* procedure in DynaDAG (North & Woodhull, 2001).

Phase	Objective	Constraint
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	$\min \sum_{e=(u,v)\in E} w(e) \mid \mathrm{X(v)} - \mathrm{X(u)}$	$X(v) \geq X(u) + \Delta(u,v)$

Where:

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

λ is level or rank assignment ($\lambda(v)$: is a layer assignment of vertex v

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

X, Y : coordinate of a vertex

Δ is minimum vertex separation. This minimum separation depends on vertex shapes

In phase 2, re-ranking, 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 7 and 8. North and Woodhull (2001) proposed to use integer network simplex solver for solving those optimization problems.

Table 7. Variables in phase 2, reranking, 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 level 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 8. Constraints in phase 2, layer assignment, 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 point out that linear network simplex does not take into account the layout stability. To compensate, North and Woodhull (2001) added variables and constraints that penalized the level assignment. In this step, the DynaDAG provides a trade-off between the 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 crossing reduction problem for constrained one-sided two-layered graph layouts will take into account the layout stability. Based on the work of Forster (2004) the proposed algorithm will explicitly include a constraint that represents the layout stability. The constrained crossing reduction problem and the work of Forster (2004) will be reviewed in following section

Phase 4 of the *Process* procedure, coordinate assignment, is not discussed in this thesis; as mentioned in the *Limitations* section in Chapter 1, this thesis considers all vertices and edges 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 similarity and differences between the DynaDAG framework and the proposed constrained graph drawing framework. Like DynaDAG, the proposed constrained graph drawing framework (CGDF) utilizes a client-server model as a communication between client and server. Another similarity between the DynaDAG and the proposed *CGDF* is the transformation of hierarchical graph drawing objectives and aesthetic constraints into equivalent optimization problems. However, the proposed *CGDF* uses different approach and technique in term of solving the optimization problems. For example, DynaDAG uses network simplex for solving optimization problems, on the other hand, our constrained graph drawing framework utilize the Sugiyama and relational database for solving optimization problems. The similarities and differences between the two frameworks are described in Table 9.

Table 9. Similarities and differences between DynaDAG and the proposed *CGDF*.

Framework	DynaDAG	CGDF
Data structure	Captures a constraint of one layer assignment	Stores vertex attributes, constraints, and snapshots of previous layouts
Framework	Client-server model	Same as DynaDAG
Heuristic	Based on Sugiyama heuristic and 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	Uses Greedy-Cycle-Removal algorithm (Eades et al., 1993) and DFS for removing cycles if applicable
Phase 2: Layer Assignment	Uses integer network simplex. To compensate for layout instability, the program adds constraints to penalize the layer assignment.	Uses a modified Coffman-Graham to assign vertices into layers
Phase 3: Crossing Reduction	Uses integer network simplex	Uses modified constrained barycenter algorithm (Forster 2004)
Phase 4:Coordinate	Uses network simplex for assigning coordinates	Out of scope of this thesis
Operations	Adds vertices, adds edges, removes vertices, removes edges	Adds vertices, adds edges, removes vertices, removes edges, adds/updates ordered 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 the work of Forster (2004) was used in our constrained crossing reduction problem.

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 the order. For example, given $u, v \in L_i : c(u, v)$ *is constrained where* $pos_i(u) < pos_i(v)$. 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 if $pos(u) < pos(v)$. Otherwise, the constraint is not satisfied (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 the vertex's barycentric value from left to right, so a vertex with a greater barycentric value will be placed to the right and the vertex with a lesser barycentric will be placed to the left. A constraint $c(u, v)$ on layer L_i is violated if the barycentric value of vertex u is greater than the barycentric value of vertex v ($b(u) > b(v)$).
2. It is also known that the greater barycentric value of vertex u indicates that 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 that more edges are to the left of the vertex than to its right, as shown in Figure 21.

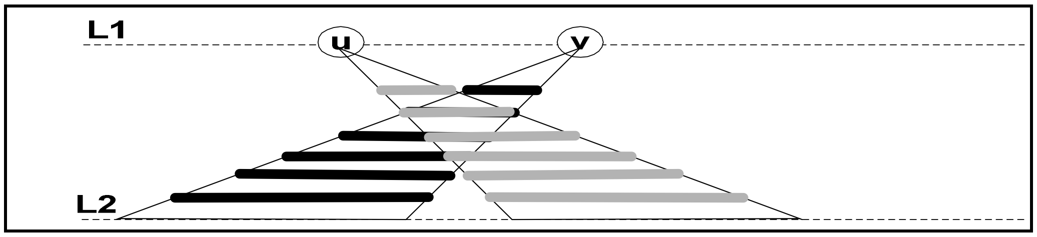
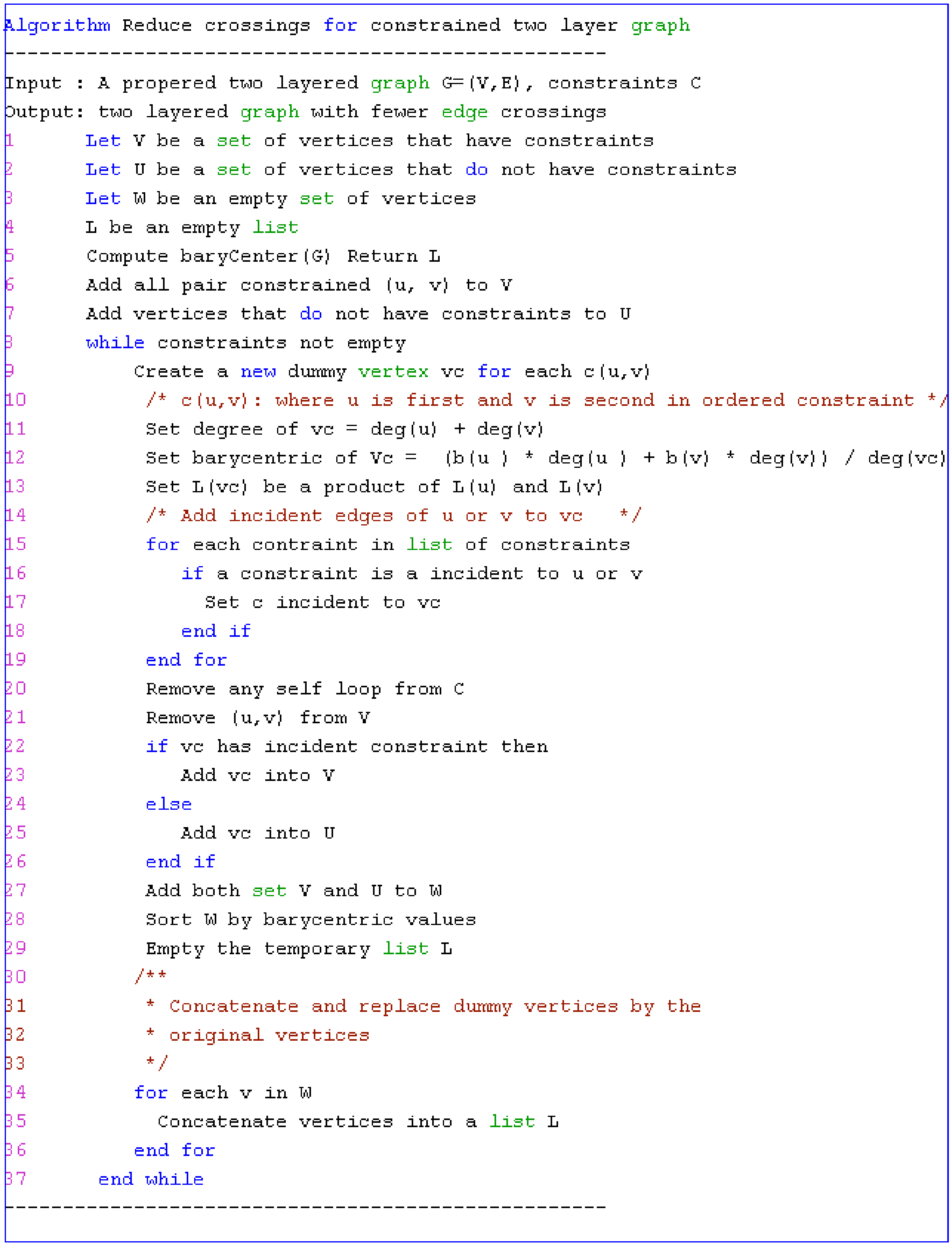


Figure 21. Barycentrics of vertices and their incident edges

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 showed that the assumption produced good layouts. The modified barycenter for constrained one-sided two-layered graphs proposed by Forster (2004) is described as follows:

Given $(G, L_{i,} L_{i+1}) \mid L_{i+1}$ is fixed. To minimize the number of crossings in layer L_i, the algorithm first calculates the barycentric values of vertices. Second, it partitions vertices into total order vertex lists.. Third, the algorithm loops through the vertices to find constraints that are violated. 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*. The barycentric value of the dummy vertex is the average value of the barycentrics of vertices *u* and *v*. This step ensures that when sorting vertices on L_i no vertex is placed between the vertices *u* and *v*, as both *u* and *v* are now temporarily 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*. Figure 22 shows the pseudocode of the modified barycenter algorithm.



```
Algorithm Reduce crossings for constrained two layer graph
---------------------------------------------------
Input : A propered two layered graph G=(V,E), constraints C
Output: two layered graph with fewer edge crossings
       Let V be a set of vertices that have constraints
       Let U be a set of vertices that do not have constraints
       Let W be an empty set of vertices
       L be an empty list
       Compute baryCenter(G) Return L
       Add all pair constrained (u, v) to V
       Add vertices that do not have constraints to U
       while constraints not empty
           Create a new dummy vertex vc for each c(u,v)
            /* c(u,v): where u is first and v is second in ordered constraint */
            Set degree of vc = deg(u) + deg(v)
            Set barycentric of Vc =  (b(u ) * deg(u ) + b(v) * deg(v)) / deg(vc)
            Set L(vc) be a product of L(u) and L(v)
            /* Add incident edges of u or v to vc   */
            for each contraint in list of constraints
               if a constraint is a incident to u or v
                 Set c incident to vc
               end if
            end for
            Remove any self loop from C
            Remove (u,v) from V
            if vc has incident constraint then
               Add vc into V
            else
               Add vc into U
            end if
            Add both set V and U to W
            Sort W by barycentric values
            Empty the temporary list L
           /**
            * Concatenate and replace dummy vertices by the
            * original vertices
            */
           for each v in W
             Concatenate vertices into a list L
           end for
        end while
---------------------------------------------------
```

Figure 22. Pseudocode of the modified barycenter algorithm (Forster, 2004)

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 work and the relationship between local drawing quality and global layout stability (Görg, 2005) influenced the design of the abstract formal model for constrained graph layout in this report. The aesthetic criteria for drawing incremental hierarchical graph layouts helped to build an abstract model for drawing comprehensible hierarchical graph layouts, and the standard Sugiyama heuristic provided a foundation for developing concrete algorithms for updating the constrained graph layouts due to dynamic operations.

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 layout used a top-down approach, as shown in Figure 23. First, an abstract model that represents incremental graph layout regardless of the family of graph layouts was designed. Next, the abstract model was adapted to represent a family of graph layouts such as hierarchical graph layout. Finally, the abstract model was transformed into concrete algorithms based on chosen algorithms. This approach enables future research to extend the work of this research by developing models for other types of graph layouts such as orthogonal, simulated annealing, etc.

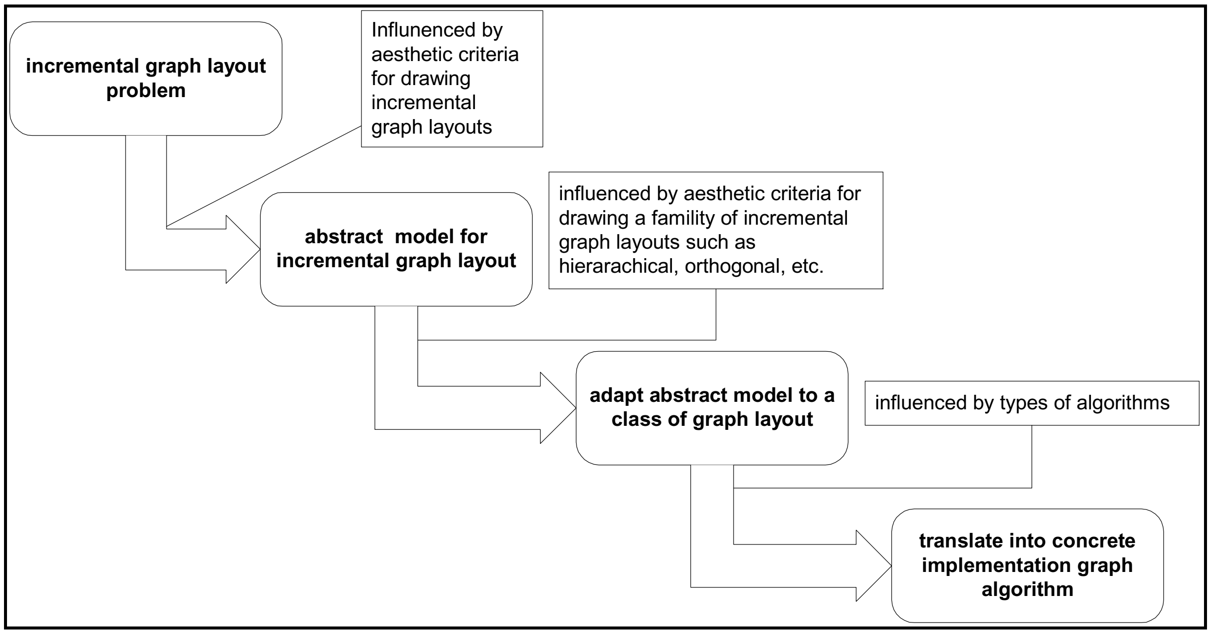

incremental graph layout problem
Influnenced by aesthetic criteria for drawing incremental graph layouts
abstract model for incremental graph layout
influenced by aesthetic criteria for drawing a famility of incremental graph layouts such as hierarachical, orthogonal, etc.
adapt abstract model to a class of graph layout
influenced by types of algorithms
translate into concrete implementation graph algorithm

Figure 23. Design flow for building an abstract model for incremental graph layouts

Details of the Abstract Model

As discussed in the section Aesthetic Criteria for Incremental Graph Layouts, the three important aesthetic criteria are consistency, layout stability, and readability. The abstract model for incremental graph layout was designed 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, the 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) shows that the readability or local layout quality conflicts with layout stability or global layout quality. Hence, in this report's abstract model the values of weights of drawing readability and layout stability were defined by end users, and the optimal solution for this problem was a balance between *local drawing quality* and *global drawing quality*. In summary, the abstract model for incremental graph layout is an optimization problem of three aesthetic criteria. Each criterion is weighted by its importance, as shown in Figure 24 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.

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

Figure 24. Description of the abstract model for constrained graph layout

This 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 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 abstract model we can define a constrained graph layout as follows:

Given a sequence of *n* graphs $g_1, g_2, \ldots, g_n$. Compute layouts $l_1, l_2, \ldots, l_n$ 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 offline 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. An abstract model for incremental graph layouts was developed 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 *multiphased*. 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 abstract model for hierarchical graph layout comprises suboptimization problems corresponding to each step in the Sugiyama heuristic.

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

The first step 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 hierarchical constrained graph drawing system uses 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 step of the Sugiyama heuristic was solved using prior knowledge stored in a relational database and that step requires no optimization problem.

The second step, 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 is the same as equation (1), as shown in Figure 24.

The third step in the Sugiyama algorithm, crossing reduction, minimizes the number of crossing by reordering vertices on a layer. This step affects readability and layout stability but not consistency, because the step 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 2. The abstract model for the third step is as follows:

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

Though the fourth step, coordinate assignment, does impact readability and layout stability, the scope of this research does not include shapes and size 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 that assigns vertices to layers using a constant value. Chapter 5 discusses the improvement of this thesis including will take into account the different sizes and shapes of vertices.

A Modified Sugiyama Heuristic for Constrained Incremental Graph Layout

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 solution for preserving the global layout stability is slightly difference from that of both offline (North & Woodhull 2001) and online (Görg 2005) approaches. DynaDAG (North & Woodhull, 2001) used a network simplex solver for solving optimization problems. To take into account the layout stability, additional constraints are added to the linear optimization problems. While the DynaDAG preserves layout stability by basing each layout solely on the previous layout. 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 drawing framework. However, Görg noted that this approach does not work automatically with multiphase algorithms like hierarchical graph layout algorithms. To accommodate the Sugiyama heuristic, Görg divides the global adjustments into multiple phases in order. The constrained graph drawing framework used the modified Sugiyama heuristic for updating the graph layouts and used a relational database to store the graph layout model and all of its snapshots. Because none of the algorithms in the standard Sugiyama heuristic take into account layout stability, The constrained incremental graph drawing framework introduced a simple solution that can be embedded within the Sugiyama algorithms to preserve stability.

To preserve the layout stability and make it easy to incorporate into multiphase algorithm like Sugiyama heuristic, the constrained graph drawing framework included an attributes called ordered constraint-*c(u,v)*. This ordered constraint is used to perverse the ordered of vertices on the same layer based on user's preference while minimizing the crossing numbers for one-sided two layer graph layout by restricting vertices from changing the order of vertices in the same layer. Additionally, based on Huang and Eades' (2005) experiment, the constrained graph drawing framework enabled end-users to change the value of *c(u, v)*. The attribute *c(u, v)* is stored in a relational database along with other vertex's attributes. The next paragraphs present the translation of the optimization problems to appropriate Sugiyama algorithms.

Step 1, cycle removal, does not impact any aesthetic criteria. Furthermore, as discussed in the previous paragraph, this step is automatically detected based on previous layouts information that are stored in the relational database and the naming convention of edge direction such that graph layout flows from top to bottom. Any cycles that may be produced in dynamic operations will be reversed while the operation is updating the layout. Thus, the modified Sugiyama heuristic does not employ the cycle removal algorithm.

Step 2, layer assignment, as discussed in the previous section, does impact all three aesthetic criteria as shown in Equation 1. Hence, the layer assignment algorithm should take into account all three criteria while reassigning vertices, which are affected by *dynamic* operations, to layers. We observe that both consistency and readability criteria are embedded in the layer assignment algorithm. For instance, assigning vertices to layers and pointing their edges in the same direction does satisfy the consistency criterion for a hierarchical graph layout. Keeping the width and height of the layout proportional satisfies the readability criterion. the constrained hierarchical graph drawing system was designed to use the modified version of the Coffman-Graham algorithm will be similar to the original algorithm but will accept a subgraph instead of the entire graph model. The impacted subgraph depends on dynamic operations which will be discussed in the architecture of the constraint graph layout framework section. For example, impacted layers due to *add vertex* operation include layers which have new vertex, any created dummy vertices, which ensures every layer is proper layer, and a set of new edges.

For step 3, crossing reduction, the constrained hierarchical graph drawing system was designed to use a modified barycenter algorithm for solving constrained crossing reduction, which extends the work of Forster (2004). The original barycenter algorithm is 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 three steps. The first step calculates barycentric values for the vertices. The second resequences vertices on the layers based on their barycentric values. The third step replaces any violated ordered constraints *c(u,v)* with a dummy vertex . Fourth, sorts vertices again based on their barycentric values. Finally, replaces the dummy vertices with real vertices.

Architecture of the Constrained Graph Drawing Framework

Similar to DynaDAG, the *CGDF* supports four basic operations and two additional operations that preserve the ordered constraints of vertices. These operations are as described as follows:

1. Add vertex—add a vertex and a set of edges
2. Add edges—add one or more edges
3. Remove a vertex and all of its incident edges
4. Remove edges
5. Add ordered constraints to vertices
6. Remove ordered constraints from vertices

The *add vertex* operation adds a vertex and a set of edges to the existing graph layout. The operation first computes the impacted layers based on a given vertex and the set of edges from the input. Next, it retrieves the existing graph layout from the database, and then inserts a new vertex and edges onto the graph layout before executing the modified Sugiyama heuristic. Finally, the operation saves the new graph layout into the database as a new snapshot. Figure 25 shows the pseudocode of the add vertex operation.

```
Algorithm Add Vertex (vertex v, set of E, Layer Li )
---------------------------------------------------
Input : vertex and set of edges
Output: reduced graph
1    Set min is the first impacted layer based on new vertex and edges
2    Set max is the last impacted layer based on new vertex and edges
3    Retrieve a reduced graph from database based on impacted layers
4    Add new vertex onto reduced graph
5    Add new edges onto reduced graph
6    Apply modified Sugiyama heuristic
7    Save the reduced graph into database as a new snapshot
---------------------------------------------------
```

Figure 25. Pseudocode of the add vertex operation

Similar to the add vertex operation, the *add edges* operation adds one or more edges to the existing graph layout. The operation first computes the impacted layers based on a given vertex and the set of edges from the input. Next, it retrieves the existing graph layout from the database, and then inserts new edges onto the graph layout before executing the modified Sugiyama heuristic. Finally, the operation saves the new graph layout into the database as a new snapshot. Figure 26 shows the pseudocode of the add edges operation.

```
Algorithm Add Edge (edges E )
---------------------------------------------------
Input : vertex and set of edges
Output: reduced graph
1    Set min is the first impacted layer based on new edges
2    Set max is the last impacted layer based on new edges
3    Retrieve a reduced graph from database based on impacted layers
4    Add new edges onto reduced graph
5    Apply modified Sugiyama heuristic
6    Save the reduced graph into database as a new snapshot
---------------------------------------------------
```

Figure 26. Pseudocode of the add edges operation

The *remove vertex* operation removes a vertex and all of its incident edges from the existing graph layout. First the operation retrieves the vertex and all of its incident edges from the graph layout from the database. Next, it computes the impacted layers based on the impacted vertex and edges. Third, the operation deletes the vertex and all of its incident edges, and then executes the modified Sugiyama heuristic. Finally, the operation saves the new graph layout into the database as a new snapshot. Figure 27 shows the pseudocode of the remove vertex operation.

```
Algorithm Delete Vertex (vertex v )
---------------------------------------------------
Input : a vertex v
Output: reduced graph
1    Delete a vertex and all its incident edges from database
2    Set min is the first impacted layer based on deleted edges
3    Set max is the last impacted layer based on deleted edges
4    Retrieve a reduced graph from database based on impacted layers
5    Run modified Sugiyama heuristic
6    Save the reduced graph into database as a new snapshot
---------------------------------------------------
```

Figure 27. Pseudocode of the remove vertex operation

The *remove edges* operation removes edges from the existing graph layout. First, the operation retrieves the edges of the graph layout from the database. Next, it computes the impacted layers based on the impacted deleted edges. Third, the operation deletes the edges, and then executes the modified Sugiyama heuristic. Finally, the operation saves the new graph layout into the database as a new snapshot. Figure 28 shows the pseudocode of the remove edges operation.

```
Algorithm Delete Edges (edges E )
---------------------------------------------------
Input : vertex and set of edges
Output: reduced graph
1    Delete edges E from database
2    Set min is the first impacted layer based on deleted edges
3    Set max is the last impacted layer based on deleted edges
4    Retrieve a reduced graph from database based on impacted layers
5    Apply modified Sugiyama heuristic
6    Save the reduced graph into database as a new snapshot
---------------------------------------------------
```

Figure 28. Pseudocode of the remove edges operation

The *add ordered constraints* operation sets ordered constraints to pairs of vertices on the same layer. The operation first retrieves the vertices' information from the database and uses the barycentric values of the vertices to determine whether any newly added ordered constraints is violated. If an ordered constraint is violated, the operation adds the violated constraint into a list of violated constraints. If the list of violated constraints is not empty, the operation executes the modified Sugiyama heuristic. Finally, it saves the latest snapshot into the database. Figure 29 shows the pseudocode of the add ordered constraints operation.

```
Algorithm Set Ordered Constraints (ordered constraint C)
---------------------------------------------------
Input : ordered constraints
Output: reduced graph
    Set min is the first impacted layer based on vertices
    Set max is the last impacted layer based on vertices
    Retrieve a reduced graph from database based on impacted layers
    Add ordered constraints onto reduced graph
    if there is violated ordered constraint
     Apply modified Sugiyama heuristic
    end if
    Save the reduced graph into database as a new snapshot
---------------------------------------------------
```

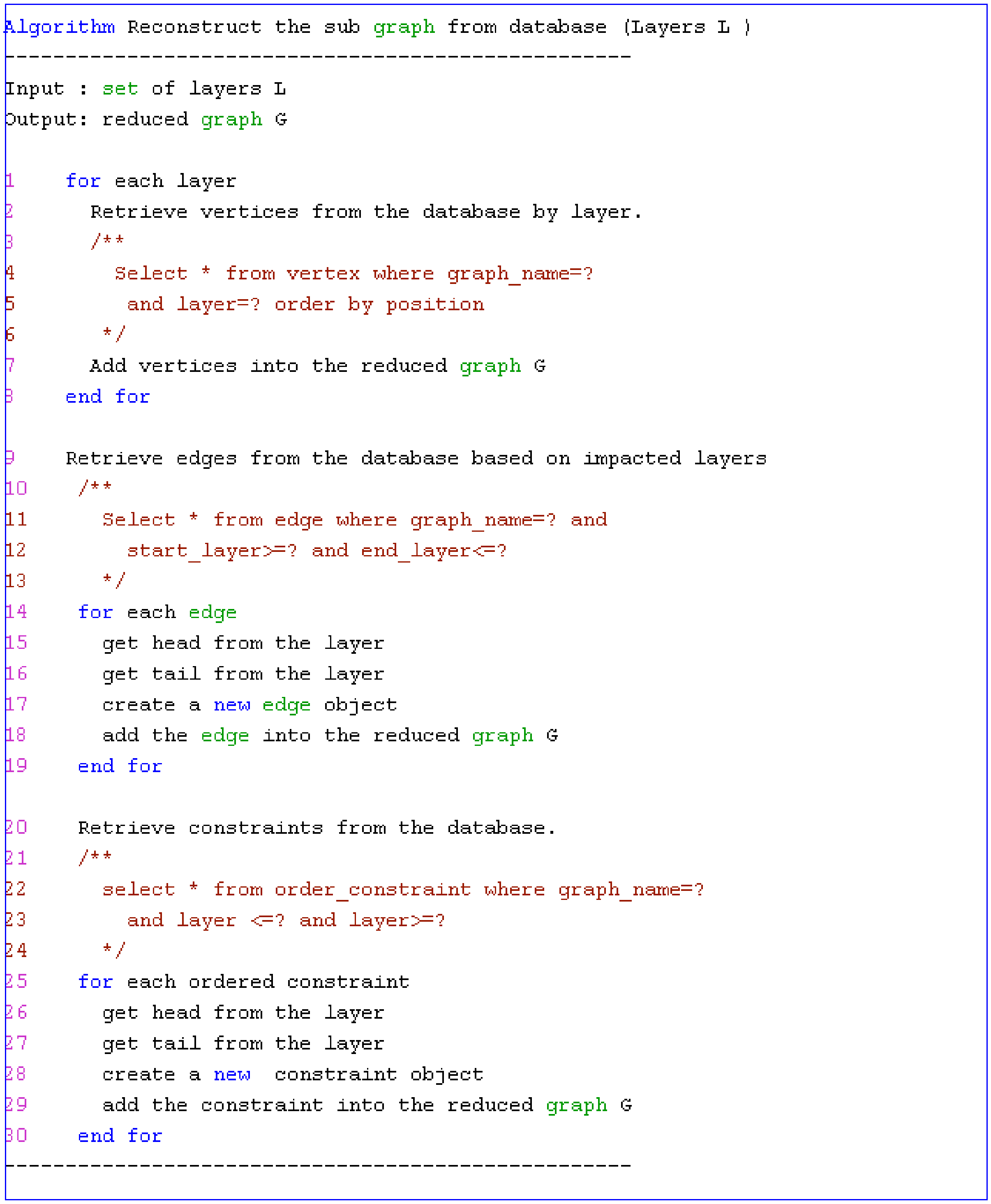
Figure 29. Pseudocode of the add ordered constraints operation

The *remove ordered constraints* operation removes ordered constraints from pairs of vertices on the same layer. The operation simply deletes ordered constraints from the database and then saves the latest snapshot into the database without executing the modified Sugiyama heuristics, as none of the aesthetic criteria are impacted. Figure 30 shows the pseudocode of the remove ordered constraints operation.

```
Algorithm Remove Ordered Constraints (ordered constraint C)
---------------------------------------------------
Input : ordered constraints
Output: reduced graph
    Set min is the first impacted layer based on vertices
    Set max is the last impacted layer based on vertices
    Delete ordered constraints from database
    Retrieve a reduced graph from database based on impacted layers
    if there is violated ordered constraint
         Apply modified Sugiyama heuristic
    end if
    Save the reduced graph into database as a new snapshot
---------------------------------------------------
```

Figure 30. Pseudocode of the remove ordered constraints operation

Each of the six operations utilizes the same function to retrieve data from the database based on the impacted layers. Figure 31 shows the pseudocode of the function that retrieves data from the database and reconstructs the subgraph that is used in each dynamic operation.



```
Algorithm Reconstruct the sub graph from database (Layers L )
---------------------------------------------------

Input : set of layers L
Output: reduced graph G

     for each layer
       Retrieve vertices from the database by layer.
       /**
         Select * from vertex where graph_name=?
          and layer=? order by position
        */
       Add vertices into the reduced graph G
     end for

     Retrieve edges from the database based on impacted layers
      /**
        Select * from edge where graph_name=? and
          start_layer>=? and end_layer<=?
        */
      for each edge
        get head from the layer
        get tail from the layer
        create a new edge object
        add the edge into the reduced graph G
      end for

      Retrieve constraints from the database.
      /**
        select * from order_constraint where graph_name=?
          and layer <=? and layer>=?
        */
      for each ordered constraint
        get head from the layer
        get tail from the layer
        create a new  constraint object
        add the constraint into the reduced graph G
      end for
---------------------------------------------------
```

Figure 31. Pseudocode of the function that retrieves data from the database to reconstruct the subgraph

To measure elegance, effectiveness, and efficiency of the proposed framework against well-known graph drawing frameworks such as Graphviz from AT&T and DynaDAG from DynaDAG.org, a simple constrained graph layout system was developed. This section describes the implementation of the constrained graph drawing framework that was used in testing. The implementation of the constrained graph drawing framework includes (1) extending the DOT language developed by AT&T, which is used to store graph layouts in the textual format; (2) developing a simple command line graph editor, which enables users to enter dynamic operations through the system console or in a file. Multiple commands can be stored in the same file and can be executed at once, which is very useful for testing; (3) a simple online graph visualization, which is an applet used to measure the graph layout elegance against the layout generated by Graphviz and DynaDAG applications; and (4) creating a relational database that stores the graph layout model and its snapshots. In addition to the implementation of the constrained graph drawing framework, the Graphviz and DynaDAG applications were installed and used as baselines for performance and elegance comparisons. The next paragraphs describe an extended version of the DOT language, the design and implementation of the constrained graph drawing framework, and the design of the entity relationship diagram.

As a large number of graph layout dataset are stored in DOT format, the constrained graph drawing framework was designed to accept input that is in the extended version of the DOT format. To simplify the implementation, the framework used the simplest version of the DOT language without including grammar for displaying shapes, descriptions, and sizes of entities and other features. Additionally, the extended version of the DOT language includes additional grammars that support six dynamic operations of the graph drawing framework. Table 10 shows the Backus Naur Form (BNF) of the extended DOT language.

Table 10. Backus Naur Form (BNF) of the extended DOT language.

```
Backus Naur Form (BNF) of an extended DOT language

graph         :     [ strict ] (graph | digraph) [ ID ] '{' stmt_list '}'
stmt_list     :     [ stmt [ ';' ] [ stmt_list ] ]
stmt          :     node_stmt
              |     edge_stmt
              |     attr_stmt
              |     ID '=' ID
              |     subgraph
attr_stmt     :     (graph | node | edge) attr_list
attr_list     :     '[' [ a_list ] ']' [ attr_list ]
a_list              :     ID [ '=' ID ] [ ',' ] [ a_list ]
edge_stmt     :     (node_id | subgraph) edgeRHS [ attr_list ]
edgeRHS       :     edgeop (node_id | subgraph) [ edgeRHS ]
node_stmt     :     node_id [ attr_list ]
node_id       :     ID [ port ]
port          :     ':' ID [ ':' compass_pt ] | ':' compass_pt
subgraph      :     [ subgraph [ ID ] ] '{' stmt_list '}'
compass_pt    :     (n | ne | e | se | s | sw | w | nw | c | _)
ID            : [-]?(.[0-9]+ | [0-9]+(.[0-9]*)? | [a-zA-Z\200-\377]
edgeop              : ->

//extension
drop_graph                        : drop ID
layer                             : [0-9]+
id_list                           : ID [; id_list]
order_list                        : ID '<' ID [ ';' order_list ]
edge_list                         : ID edgeop ID [; edge_list ]
add_vertices                      : add vertices ID ID layer '{' edge_list '}'
remove_vertices                   : remove vertices ID ID
add_edges                         : add edges ID '{' edge_list '}'
remove_edges                      : remove edges ID '{' edge_list '}'
set_order_constraints             : set order ID '{' order_list  '}'
remove_order_constraints          : drop order ID '{' order_list '}'
```

Drop_graph grammar is used to delete a graph layout and all its snapshots from the database. *Layer* grammar represents layers of a graph layout. *Order_list* grammar represents a list of ordered constraints, which is a pair of vertices on the same layer. Other grammars are self-explanatory. An example of the extended DOT language used in testing is shown in Table 11.

Table 11. An example of extended DOT language.

```
add vertices "/home/mvinni/o/jspin411/tmp_t/thirdabbrev" 50 1 { 50->0 }
add edges "/home/mvinni/o/jspin411/tmp_t/thirdabbrev" { 50->4 }
remove vertices "/home/mvinni/o/jspin411/tmp_t/thirdabbrev" 2
set order "org-parent-child-conversion-12-134" { 8482 < 8479  }
```

The *CGDF* system's architecture is similar to that of the DynaDAG (North & Woodhull 2001). It is a client-server application that uses TCP and HTTP protocols to communicate between the clients and server. Once the users execute the instructions using the editing tool, the client sends the command to the server. The server then updates the graph layout based on the *dynamic* operations by executing the Sugiyama heuristic, and then stores a new snapshot of the layout in a relational database. The client includes a drawing editing tool and a graph layout visualization component. The high-level architecture of the developed system is shown in Figure 32.

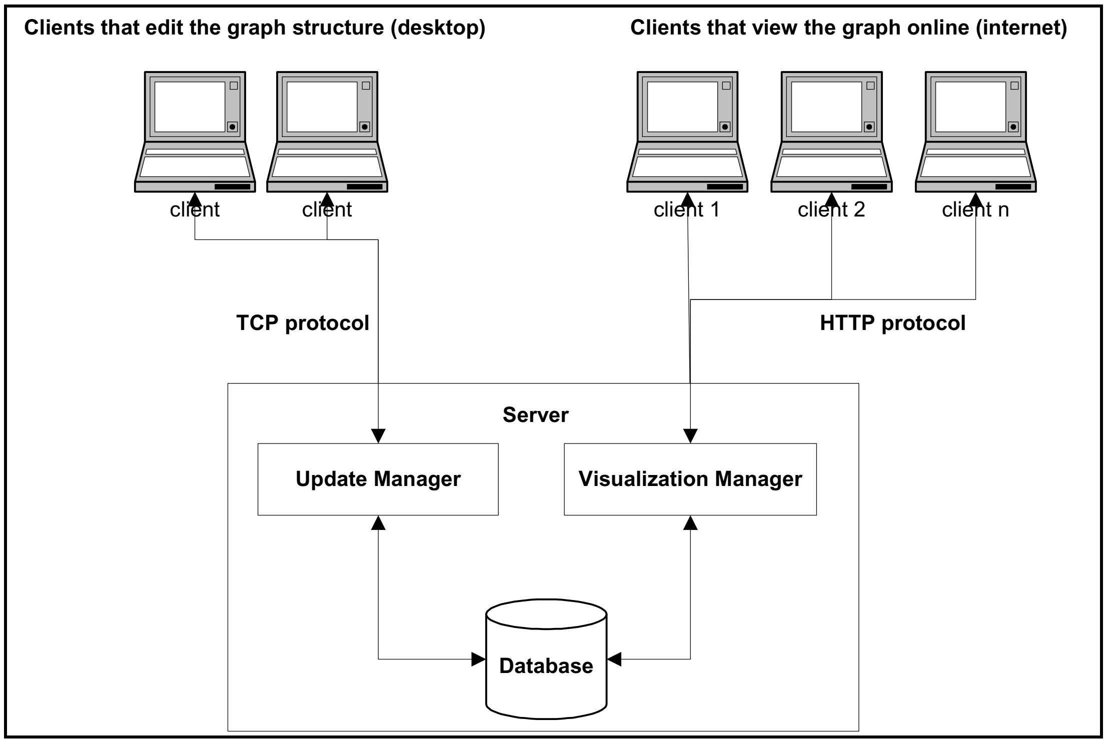


Figure 32. Architecture of the graph drawing framework

The graph editor, which runs on a desktop computer, is a simple command line editor that accepts input from a command line. The input can be entered from either the console or a file. To support automatic testing, the graph editor accepts one-to-many dynamic operations from the same file. Once the input is entered, the editor parses the input and invokes an appropriate action, which in turn calculates the layers that are impacted by the dynamic operation, and then retrieves data and executes the modified Sugiyama heuristics. Finally, the action saves the snapshots into the database. The execution flow of the graph editor component is shown in Figure 33.

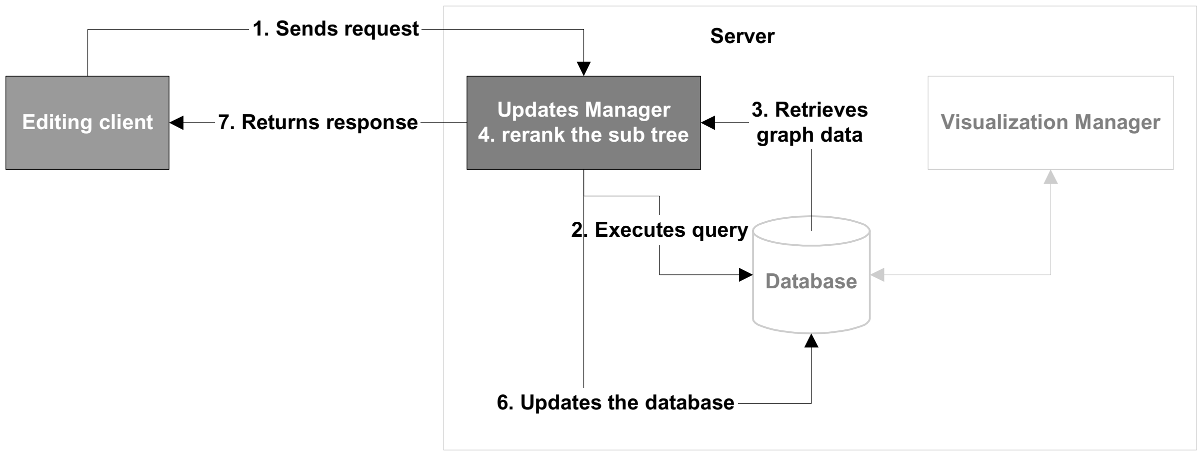


Figure 33. Execution flow of the graph editor component

The graph visualization component was developed as an applet that can be run on a desktop or the Internet. Due to the limitation of the thesis, which does not include Step 4 in the Sugiyama heuristic that calculates the horizontal coordinate assignment for vertices, the graph visualization rendering engine simply uses the positions of vertices on a layer, which was calculated in the crossing reduction step of the Sugiyama heuristic, to render the vertices and edges. As a result, the generated layouts do not look as pleasant as the layouts generated by Graphviz. The visualization test scores the developed graph visualization component based on the number of vertices on a layer and the number of edge crossings. The applet was designed to be an interactive application that enables end users to select a graph layout to be displayed. Once a layout is selected the applet retrieves all the snapshots of the layout from the database through a web service call. The applet then renders the graph layout snapshot. The execution flow of the graph visualization component is shown in Figure 34.

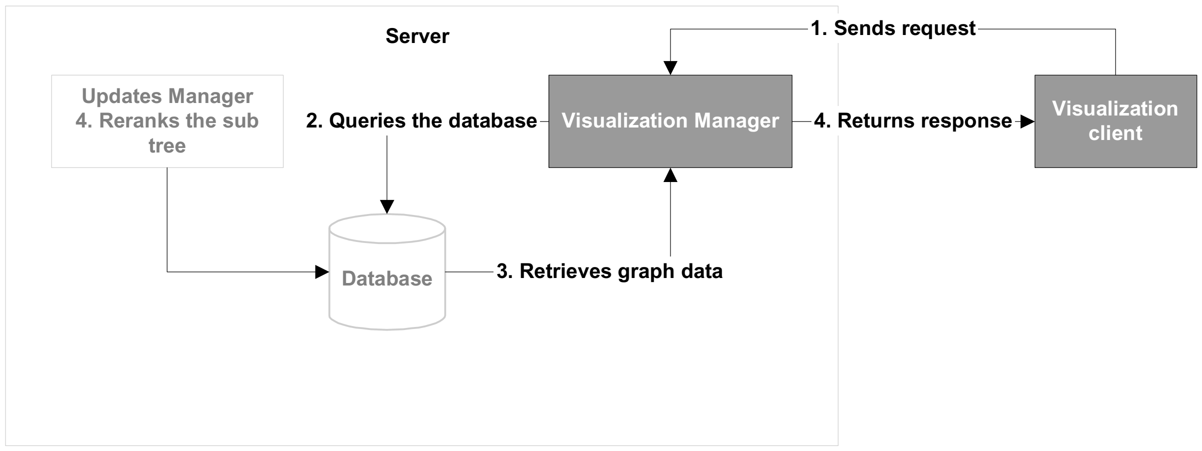


Figure 34. Execution flow of the graph visualization component

The developed graph editor and visualization system was written in Java. The program comprises eight main packages, as shown in Figure 35. The packages lex, parser, and absyntax were used to parse input from the DOT data file into abstract syntax. The action package is the main driver that translates the abstract syntax into objects, executes a modified Sugiyama heuristic, and saves objects into a relational database through the use of a data object package. The algorithm package contains classes that implement algorithms in the Sugiyama heuristic. The plus sign notation used in the UML diagrams represents the composition relationship.

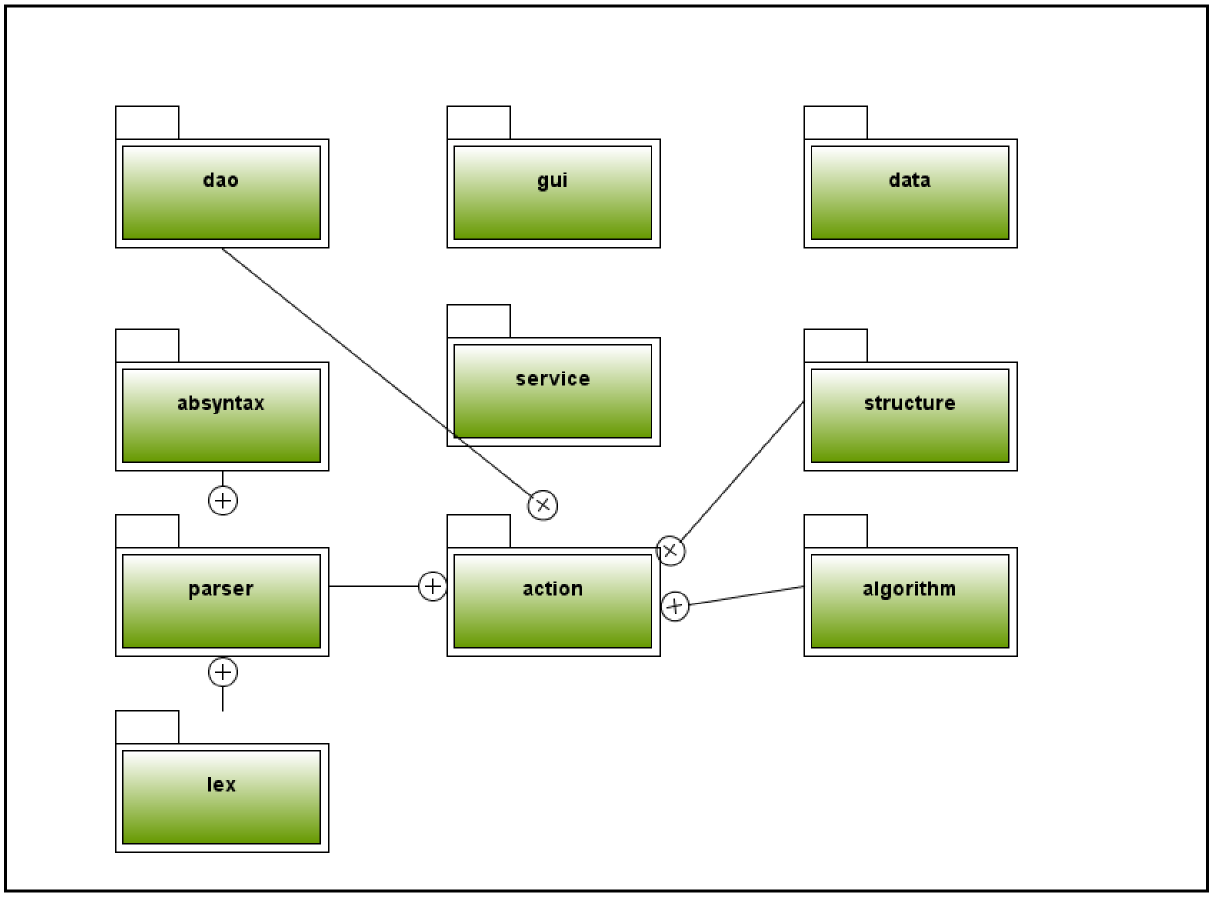


Figure 35. Packages of the constrained graph layout application

This section describes the design and class diagrams of the main classes of the constrained graph drawing framework. Other utilities and helper classes, along with the source code of the Sugiyama algorithms, are shown in Appendix A. Since the algorithm package contains all the algorithms in the Sugiyama heuristic, the class diagrams of this package are divided into groups based on their functionality. Figure 36 shows the class diagram of the cycle removal algorithm. The Factory pattern design was utilized to enable the extendibility of the program. For example, additional implementations of the sorting algorithms could be implemented without changing the abstract layer of the main algorithm. The factory also enables the system to instantiate different concrete implementations of the algorithm in real time. The main class GreedyCycleRemoval calls the sorting algorithm through the use of the Factory class. The sorting algorithms are self-explanatory based on their names.

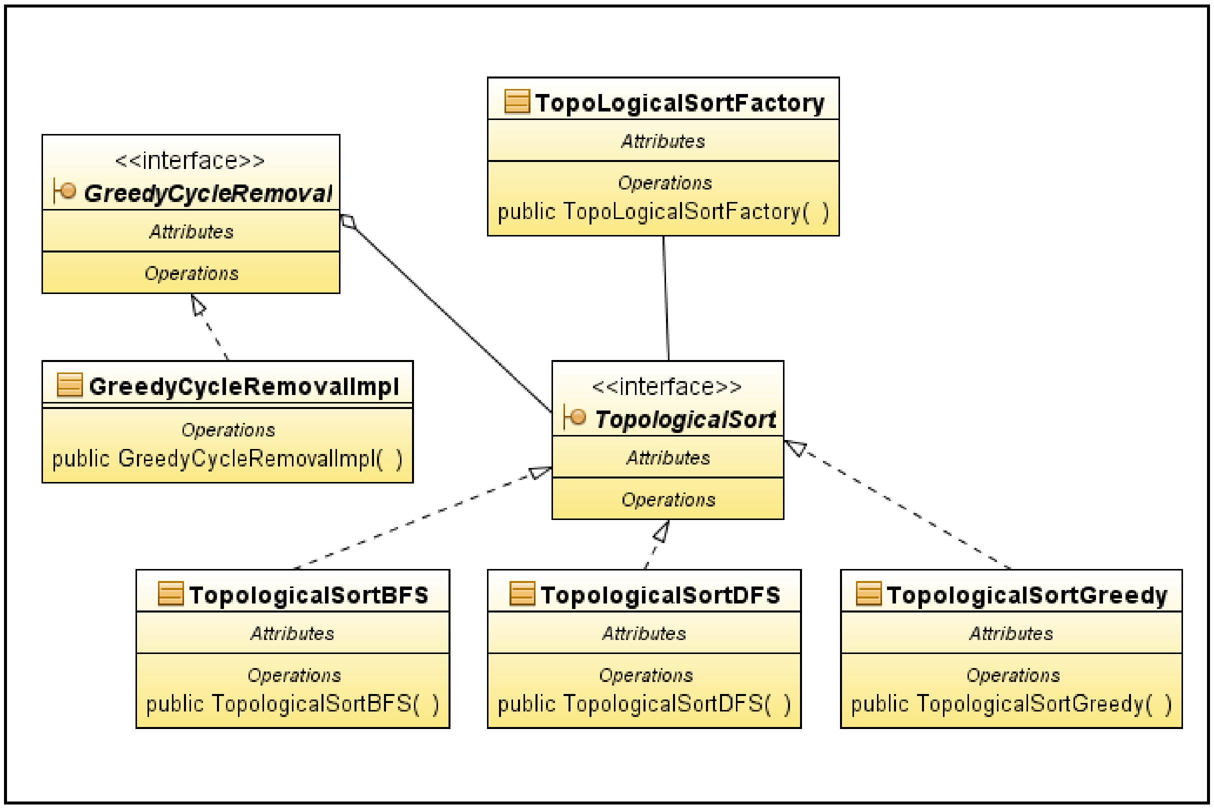


Figure 36. Class diagram of the cycle removal algorithm

The second group of the algorithm in the Sugiyama heuristic is the layer assignment algorithm. In the same manner, the factory pattern was also utilized in the layer assignment algorithm, as shown in Figure 37. The Layer Assignment class is called through a Factory class that can instantiate different concrete implementations of the layer assignment algorithm.

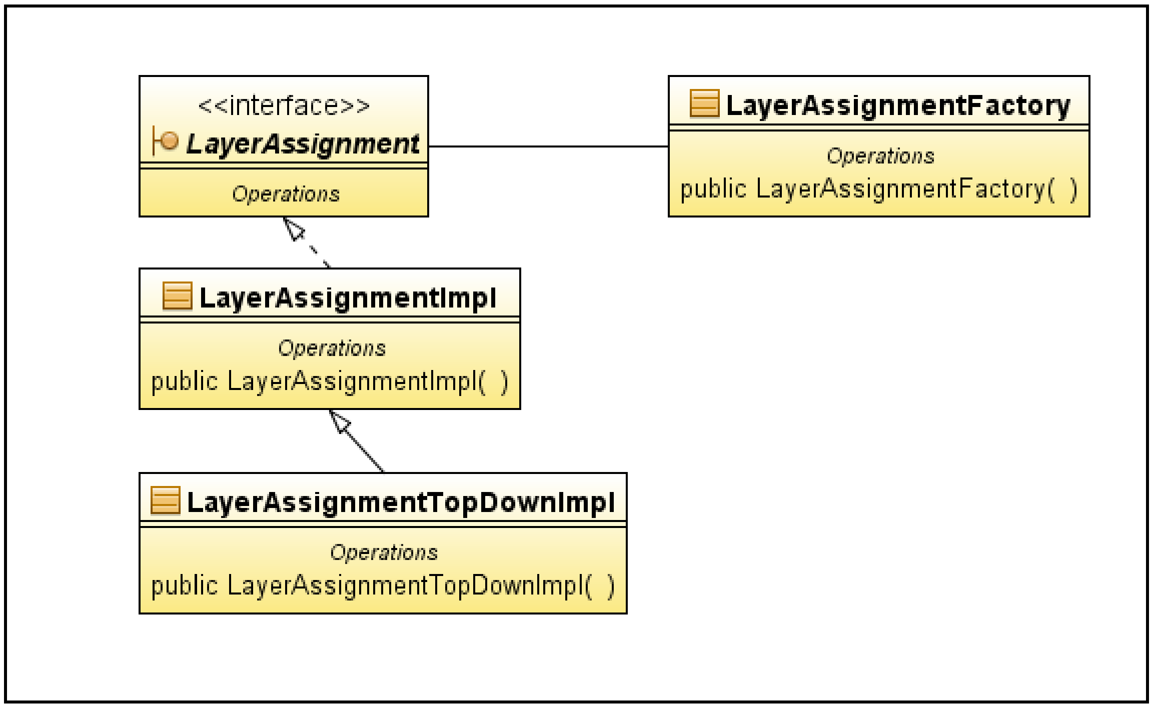

<<interface>>
LayerAssignment
Operations
LayerAssignmentFactory
Operations
public LayerAssignmentFactory( )
LayerAssignmentImpl
Operations
public LayerAssignmentImpl( )
LayerAssignmentTopDownImpl
Operations
public LayerAssignmentTopDownImpl( )

Figure 37. Class diagram of the layer assignment algorithm

Crossing reduction algorithms are shown in Figure 38. The crossing sweeping layer-by-layer class calls a two-layer crossing reduction algorithm through a factory class. The SweepingReductionStatic class was used to compute the number of crossings for the static graph, and the SweepingReductionDynamic class was used to compute the crossing number for layouts due to dynamic operations. In the same manner, to compute the crossing numbers for a two-layer graph layout, two two-layer crossing reduction algorithms were implemented. The CrossingReductionBaryCenter class was designed to reduce the crossing number of the static graph or initial graph, and the CrossingReductionConstrainedBaryCenter class was designed to reducing the number of crossings for layouts due to dynamic operations.

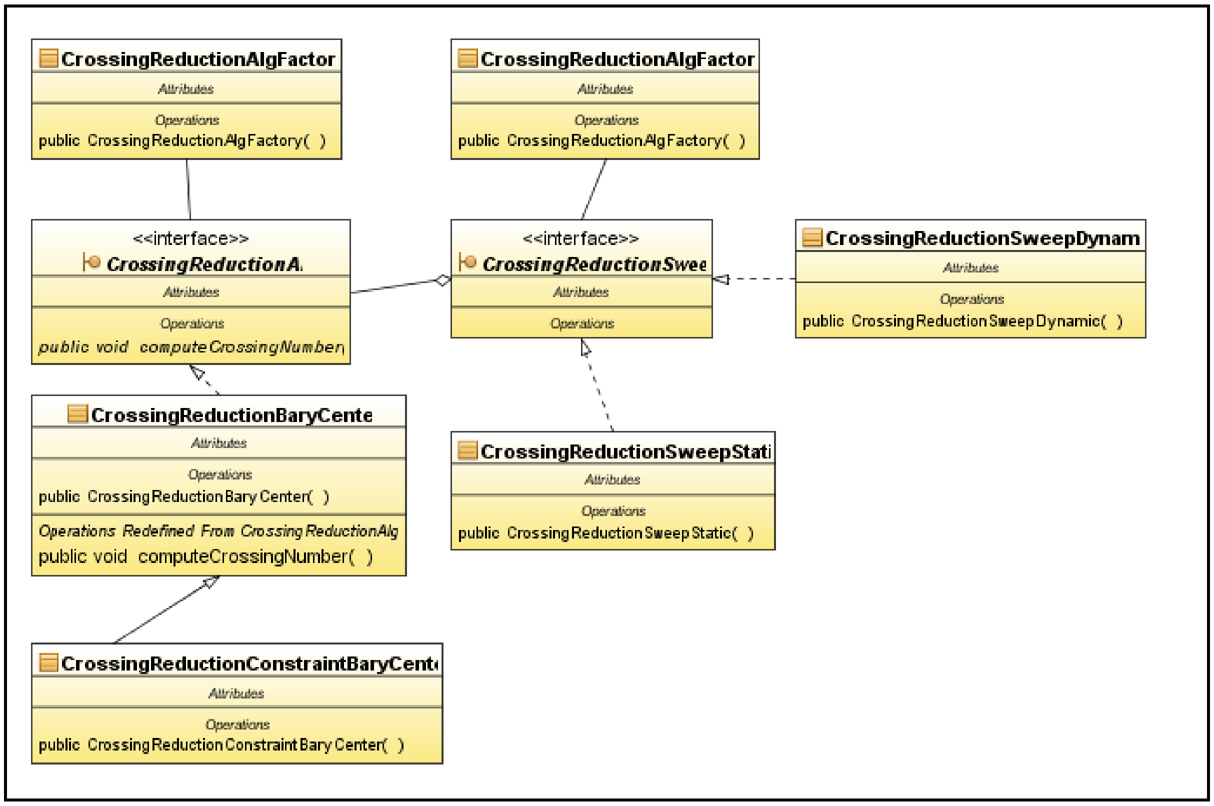

CrossingReductionAlgFactor
Attributes
Operations
public CrossingReductionAlgFactory( )
CrossingReductionAlgFactor
Attributes
Operations
public CrossingReductionAlgFactory( )
<<interface>>
CrossingReductionA
Attributes
Operations
public void computeCrossingNumber
<<interface>>
CrossingReductionSwee
Attributes
Operations
CrossingReductionSweepDynam
Attributes
Operations
public CrossingReductionSweepDynamic( )
CrossingReductionBaryCente
Attributes
Operations
public CrossingReductionBaryCenter( )
Operations Redefined From CrossingReductionAlg
public void computeCrossingNumber( )
CrossingReductionSweepStati
Attributes
Operations
public CrossingReductionSweepStatic( )
CrossingReductionConstraintBaryCent
Attributes
Operations
public CrossingReductionConstraintBaryCenter( )

Figure 38. Class diagram of the crossing reduction algorithm

The design of package *Action* was simple because it contains one interface named Action and a main class called GraphAction, which implements the Action interface, as shown in Figure 39. This package is the main engine of the graph editor component. GraphAction was designed as a command-line editor that accepts input from either a console or a file. This class will then invoke appropriate action based on the input. It then retrieves a subgraph from the database, executes the modified Sugiyama algorithms if applicable, and finally updates the database with a new snapshot. The package also contains most of the test classes that were used to compare performances and elegance against Graphviz and DynaDAG applications. The test source code is shown in Appendix B.

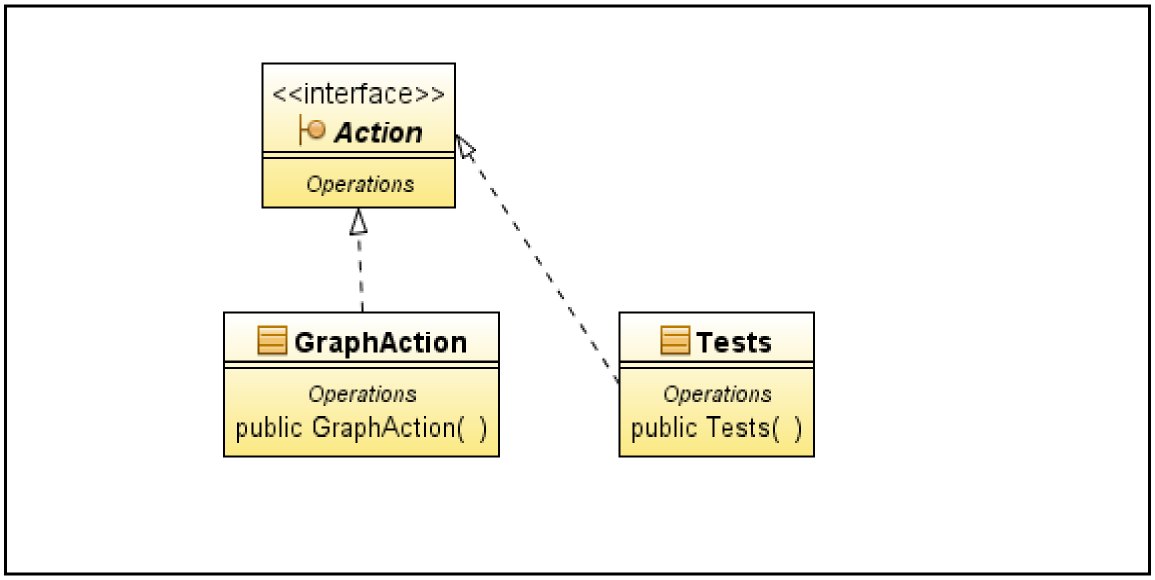


Figure 39. Class diagram of the action package

The package *structure* was designed to capture the graph model from the DOT format and from a relational database. The package contains a main class called SimpleGraph, which was designed to store a graph layout in memory. In this class it contains one-to-many instances of Vertex and Edge classes. In addition to those main classes, the LayerDataStructure class was designed as a helper class for the layer assignment step. This class helps to keep track of the layers of the graph layout. The class diagram is shown in Figure 40.

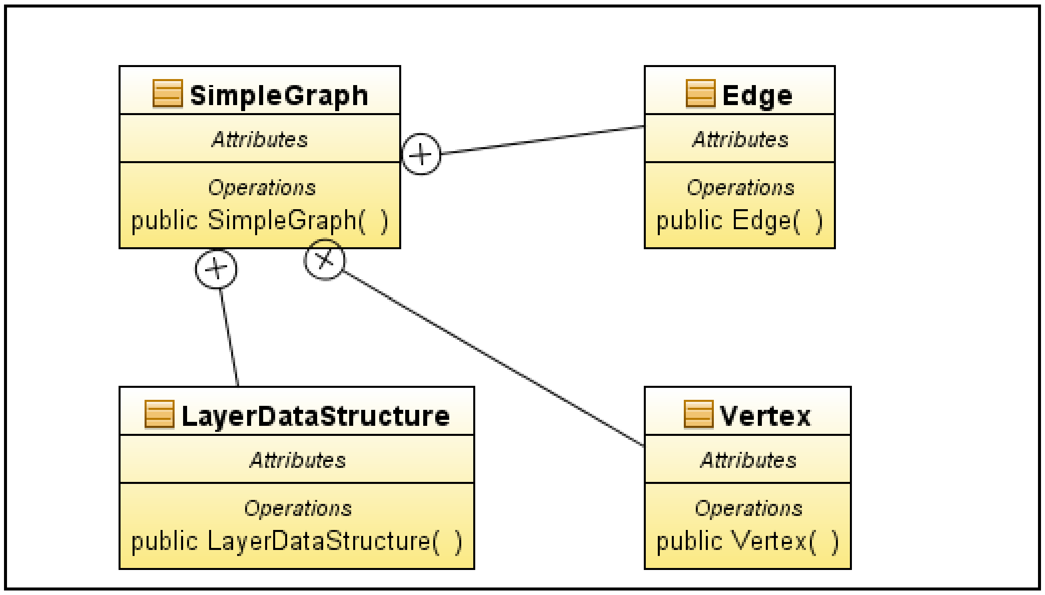


Figure 40. Class diagram of the structure package

As parser, lexical, and absyntax packages were designed to parse graph layouts from the DOT into memory, all three class diagrams of these three packages are shown in Figure 41. The Parser was designed to translate graph layouts from the DOT language into appropriate abstract syntax types. Different types of abstract syntax classes help the GraphAction class determine the appropriate action to be called.

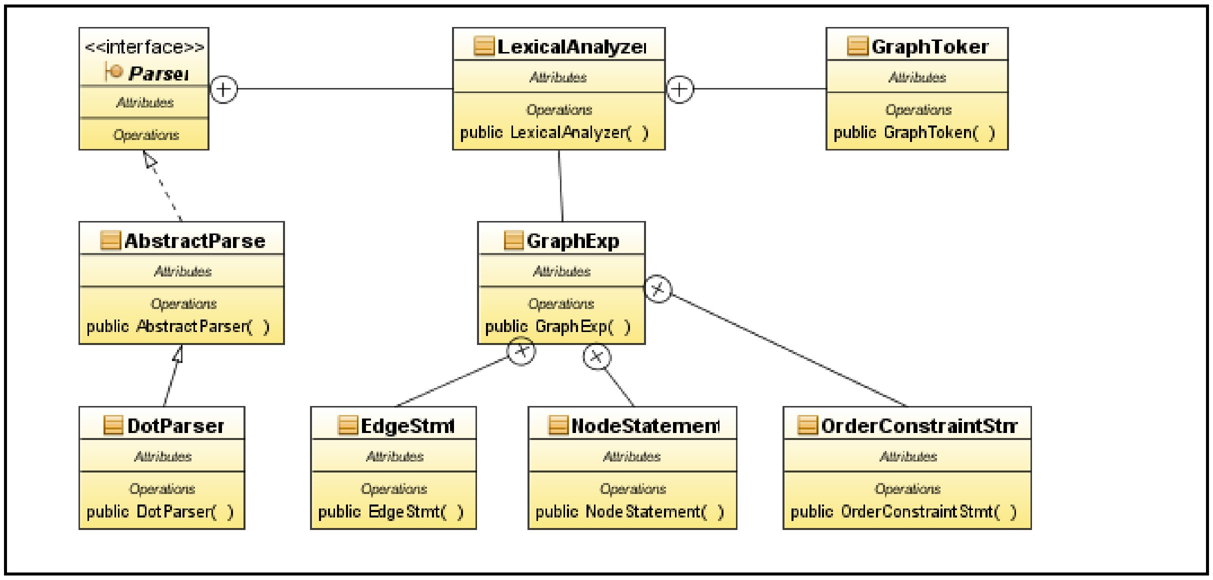


Figure 41. Class diagram of the parse, lexical analysis, and abstract syntax classes

Package *dao* was designed to map data from the relational database to objects, and the *service* package was designed to transfer the data between client and server through the HTTP protocol. Once it receives the HTTP request from the client, the service package invokes the classes in package dao to retrieve data from the relational database. The class diagram of the dao and service packages is shown in Figure 42.

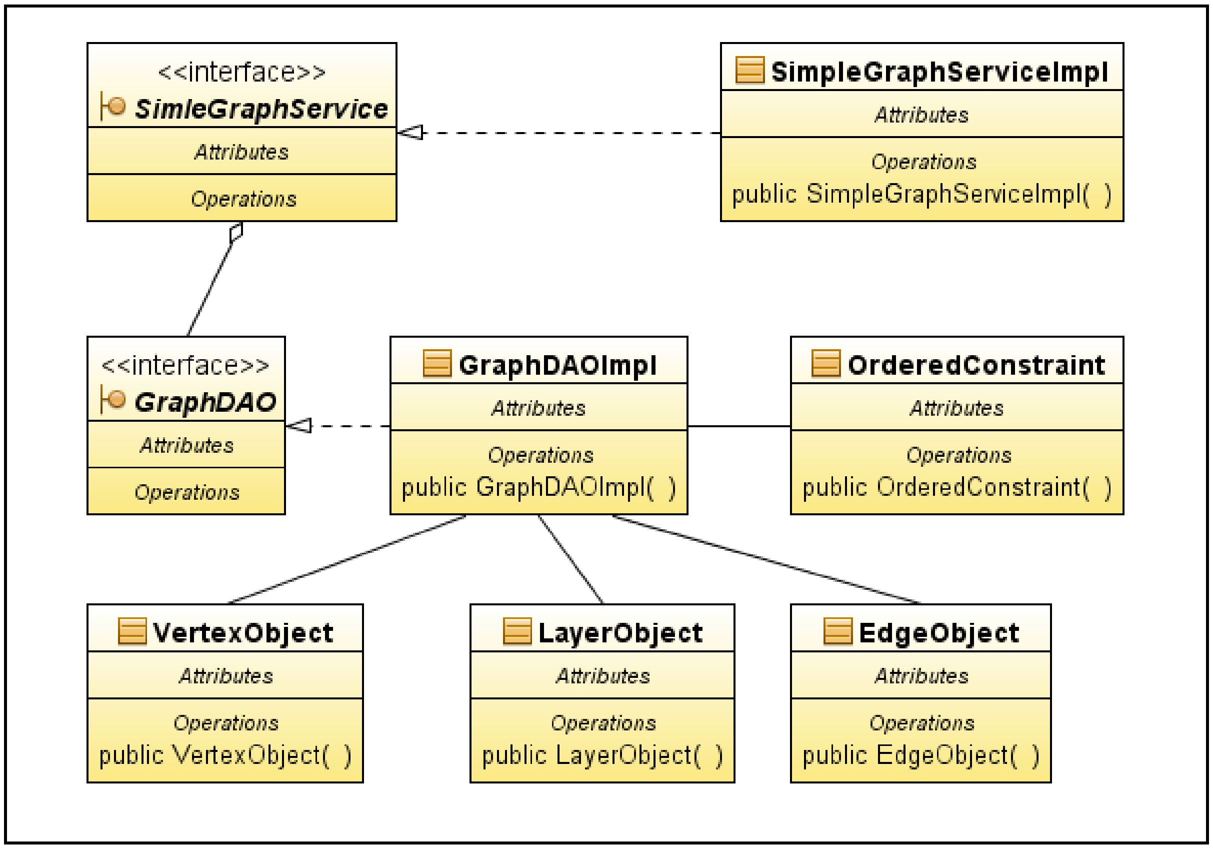


Figure 42. Class diagram of the dao and service packages

Package *gui* contains the main classes for displaying the graph layout, and the data package was designed to map vertex and edge objects into graphical shapes such as circles, lines, and rectangles. Once the SimpleGraphServiceImpl class has completed data retrieval from the relational database, the classes in the data package are instantiated and transform vertices and edges into circle and rectangle shapes. The coordinates of vertices and edges are simply calculated based on the positions of vertices on the layers. These positions are stored in a database. The dao package was designed to map data from the relational database to objects. The class diagram of gui, data, and geom packages is shown in Figure 43.

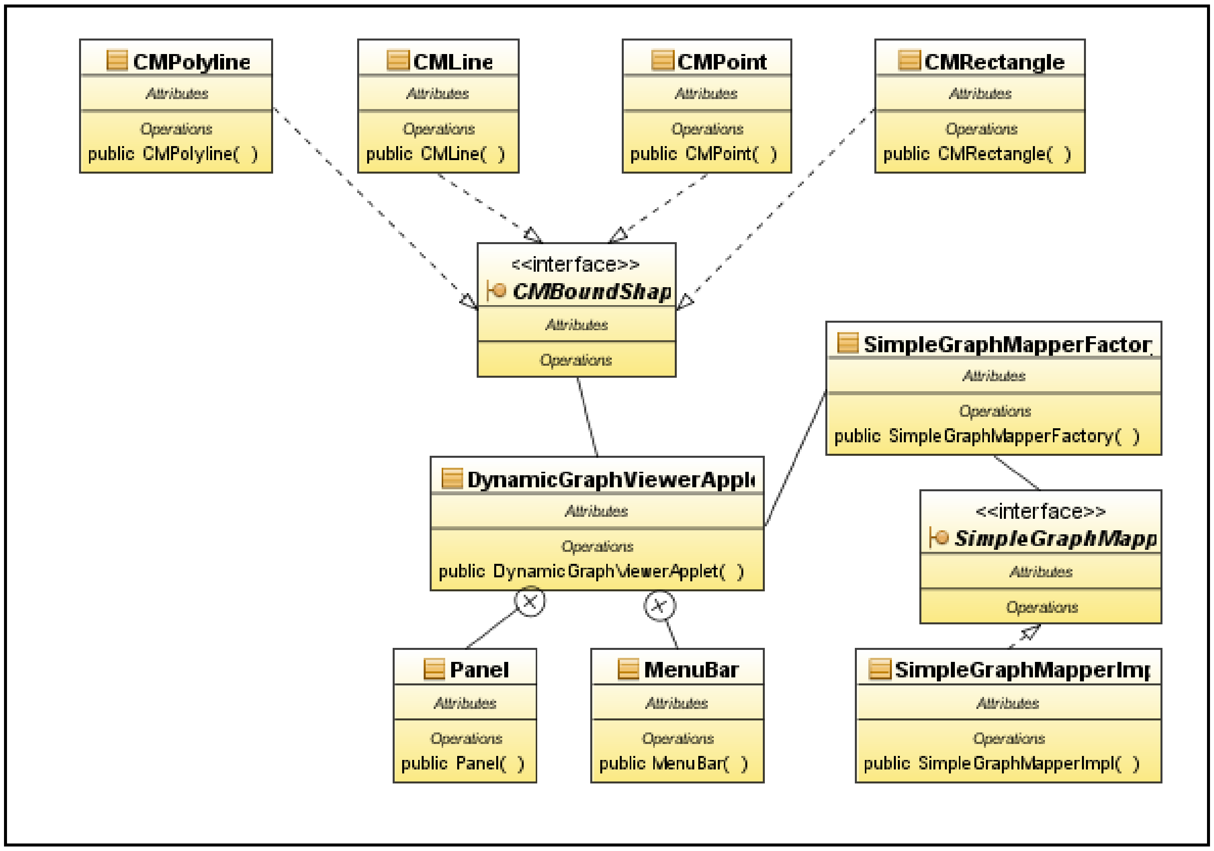


Figure 43. Class diagram of the main classes in gui, data, and geom packages

An Entity Relationship for Constrained Hierarchical Graph Drawing

To enable the drawing visualization to render graphs with thousands of vertices, the constrained hierarchical graph drawing framework utilized a relational database to capture the graph model and a sequence of the graph layouts to speed up graph visualization.

Computing hardware has been progressing rapidly in the past decade, especially in the data storage field, so our design took 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. Furthermore, because the cost of joining tables is expensive in terms of performance, the database structure was designed to increase space complexity but decrease time complexity by ensuring that data retrieval can be achieved without joining multiple tables. This was done by storing snapshots of vertices and edges in separate tables without sharing common vertices, edges, or constraints with the vertex, edge, and constraint tables respectively. Thus, a relational database was utilized to store snapshots of each layout in the graph layout sequence. The entity relationship diagram for the model is shown in Figure 44.

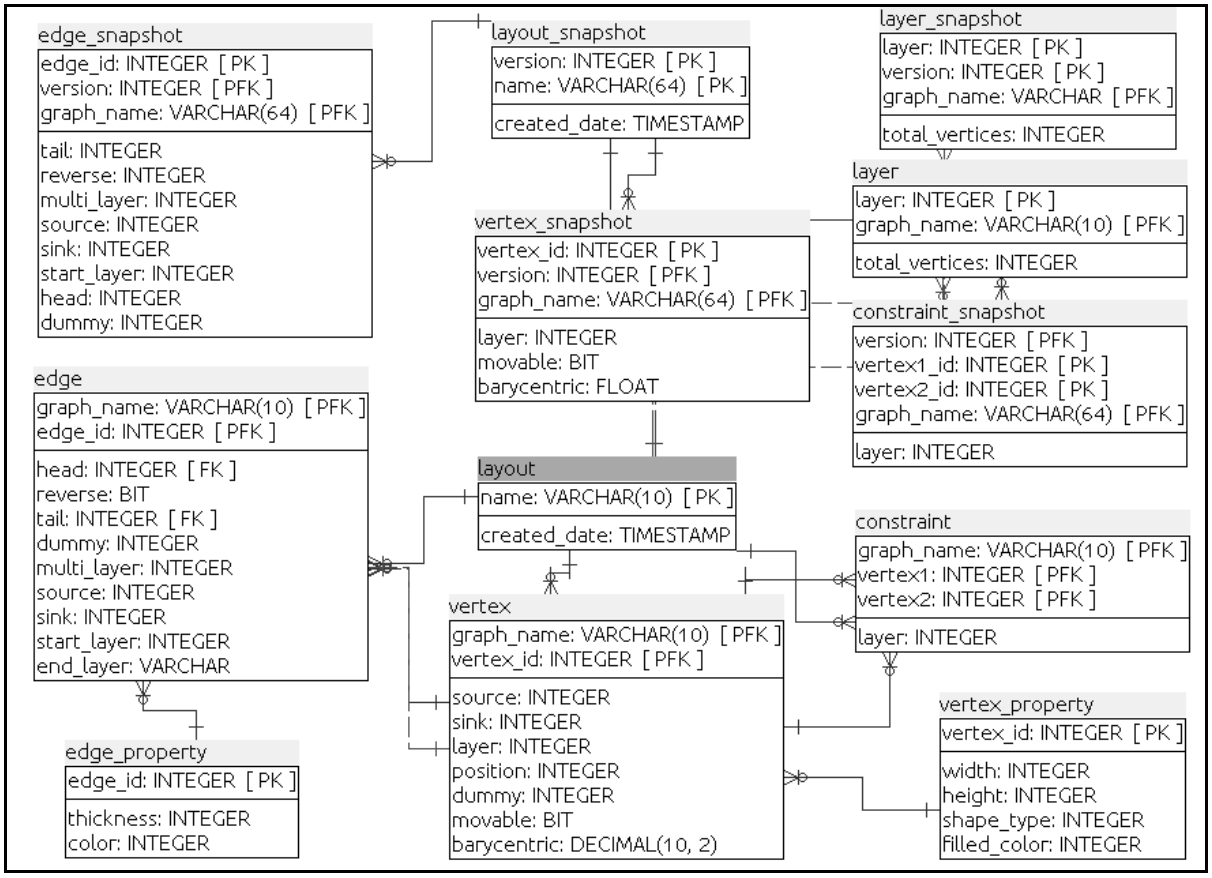


Figure 44. Entity relationship diagram for the constrained graph drawing framework

As shown in Figure 44, the entity relationship diagram comprises 10 core relations that store the graph layout model and graph layout snapshots. The relations *layout*, *layer*, *vertex*, *edge*, and *constraint* capture the up-to-date graph layouts. The relations *layout_snapshot*, *layer_snapshot*, *vertex_snapshot*, and *edge_snapshot* capture snapshots of graph layouts. The current implementation of the constrained graph framework was not designed to support different type of shapes and colors for vertices and edges, however our data model was designed to include additional properties such as vertex size and shape, and edge colors. The entity relationship includes two additional relations that store entity properties. The additional designed data structure supports future enhancement of the constrained graph drawing framework. Thus they are not used in the current implementation.

To make relation naming consistent, the relations that store snapshots of the layout were named such that their prefix is the name of the relation that stores the model; the suffix was called *snapshot*. For example, the *layout* relation stores the model of a layout, and the *layout_snapshot* relation stores the snapshots of the layout. From this point forward, the relations that store actual models of graph layouts will be called *model* relations and the relations that store snapshots of layouts will be called *snapshot* relations. Furthermore, both *model* and *snapshot* relations have the same number of attributes, with the exception that the snapshot relation has an additional attribute called *version*, which enables it to store multiple versions or snapshots of the same layout. The following description tables show both model and snapshot attributes.

Table 12. Relations *layout* and *layout snapshot*.

Attribute	Description	Purpose
name	Name of graph layout	Enables storage of one-to-many layouts
created_date	Creation date	For tracking purposes
version	Version of snapshot	Enables storage of one-to-many layout snapshots

Table 13. Relations *layer* and *layer snapshot*.

Attribute	Description	Purpose
name	Name of graph layout	Enables storage of one-to-many layouts
layer	Layer number	Preserves the layer assignment step in the database
total-vertices	Numbers of vertices on the layer	Helper attribute in graph visualization
version	Version of snapshot	Enables storage of one-to-many layout snapshots

Table 14. Relations *vertex* and *vertex_snapshot*.

Attribute	Description	Purpose
graph_name	Name of the graph layout	graph_name and vertex_id attributes form a primary key of the vertex table
layer	Layer vertex resides on	Preserves layer assignment step in the database
movable	Movable constraint	Restricts vertex from moving while computing crossing reduction
barycentric	Barycentric value of vertex	Preserves crossing reduction step in the database
position	Position of vertex on a layer	Preserves crossing reduction step in the database
vertex_id	Id of a vertex	graph_name and vertex_id attributes form a primary key of the vertex table
dummy	If this attribute = 1 then vertex is dummy. Otherwise, vertex is real	Keeps track of created dummy vertices in the layer assignment step
source	If this attribute = 1 then vertex is a source of a multilayer edge	Helper attribute that keeps track of multilayer edges
sink	If this attribute = 1 then vertex is a sink of a multilayer edge	Helper attribute that keeps track of multilayer edges
version	Snapshot version	Supports multiple snapshots of the same layout

Table 15. Relations *edge* and *edge_snapshot.*

Attribute	Description	Purpose
graph_name	Name of the graph layout	Distinguishes vertex from one layout to another
head	Source vertex	Source of an edge
tail	Sink vertex	Sink of an edge
reverse	If reverse = 1 then the edge is in cycle	Preserves the original direction of an edge
multi_layer	If reverse = 1 then the edge is a multilayer	Keeps track of multilayer edges
start_layer	First vertex in multiedge	Helps to track all dummy vertices and edges
end_layer	Last vertex in multiedge	Helps to track all dummy vertices and edges
dummy	If this attribute=1 then edge is dummy.	Keeps track of created dummy edges in the layer assignment step
source	If this attribute=1 then vertex is a source of a multilayer edge	Helper attribute that keeps track of multilayer edges
sink	If this attribute=1 then vertex is a sink of a multilayer edge	Helper attribute that keeps track of multilayer edges
version	Snapshot version	Supports multiple snapshots of the same layout

Table 16. Relations *order_constraint* and *ordered_constraint snapshot.*

Attribute	Description	Purpose
graph_name	Name of graph layout	Enables storage of one-to-many layouts
vertex_id_1	First vertex	First vertex in ordered constraint
vertex_id_2	Second vertex	Second vertex in ordered constraint
layer	Layer where vertices resides	Helper attribute in the crossing reduction algorithm
version	Version of snapshot	Enables storage of one-to-many layout snapshots

The Process of Collecting Graph Data

We collected graph data from the following sources:

- The DOT data file from the Graphviz repository
- Real-world graph data of organizational hierarchical relationships and enterprise processes from The Boeing Company. The process graph data has about 3,000 vertices and the organizational graph data has about 10,000 vertices. These two graph datasets were then used to generate a pool of graph datasets with different sizes. These generated graph dataset were used for visualization and performance tests.
- Real-world large graph data of CAIDA AS (autonomous systems) from the Stanford large-graph data repository.

Testing and Evaluation

We employed suggestions by Eades (2005) to evaluate and measure the constrained graph drawing framework and its constrained crossing reduction algorithm. The constrained graph drawing framework and algorithms were evaluated and measured based on the three goals shown in Figure 45. Effectiveness measures how well the framework produces layouts. Efficiency measures the performance of the algorithm and the scalability of the proposed framework, such as how well the system handles large graph layouts. Elegance measures the degree of compliance of the generated layout with graph layout aesthetic criteria.

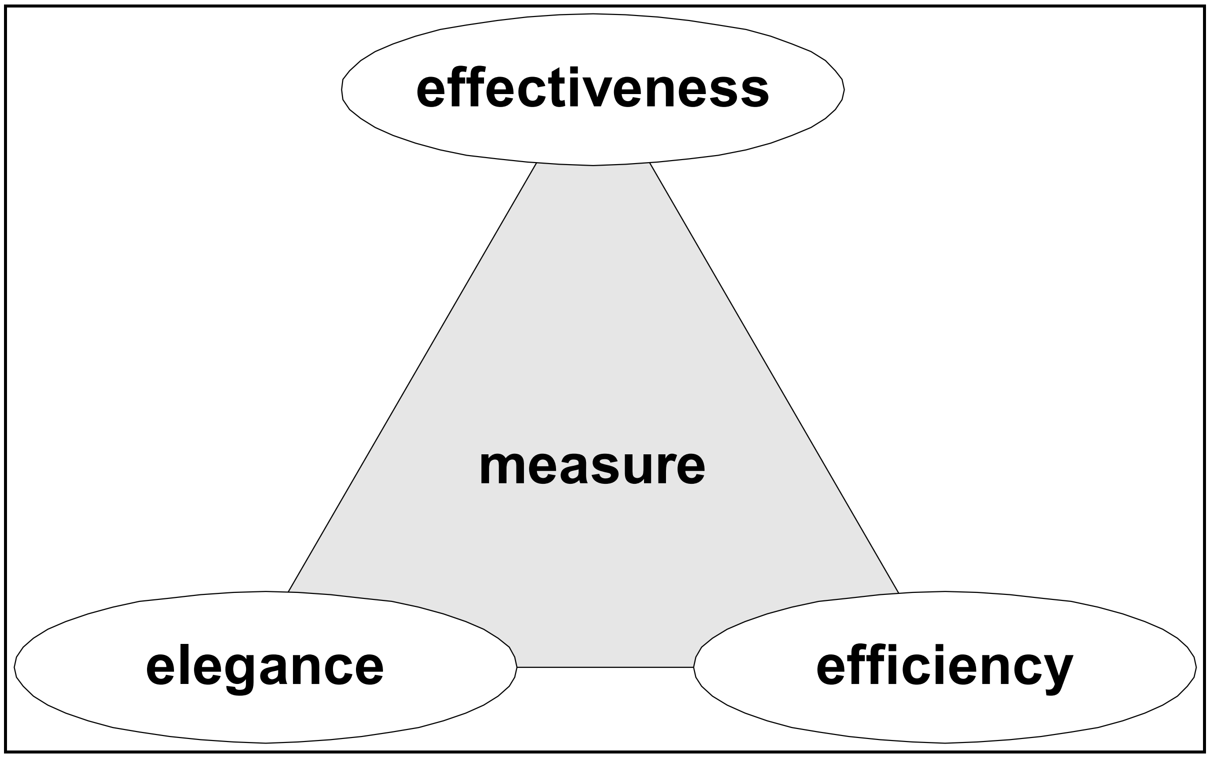


Figure 45. Measurable goals for evaluating the proposed algorithm.

To satisfy the three measurable goals, three tests were performed. Two visualization tests were performed to measure the effectiveness and elegance of the constrained graph drawing framework. The third test, which comprised a series of performance tests, was conducted to measure the performance and scalability of the constrained graph drawing framework. All tests comprised four factors: (1) setup, (2) test procedure, (3) evaluation criteria, and (4) dataset. While the test procedure and dataset of each test were different from one another, most tests used the same setup and evaluation criteria. The test procedure and datasets are discussed individually in later sections of this report. The setup and evaluation criteria are discussed in this section. In additional to these three tests, an asymptotic analysis for the modified Sugiyama algorithms is also shown.

Test Setup

For all comparison tests except the comparison of the generation of very large graph layouts, the setup comprises two test programs. The first test program, written in Java, loads the dataset into memory, executes the action procedure, which in turn computes the Sugiyama heuristic run time, and finally computes the run time. The second test program, written in Java, is a shell program that invokes the DOT program that generates the graph layout. The shell Java program computes the time it takes to complete the execution of the DOT program.

Evaluation Criteria

The evaluation criteria for performance measurement is straightforward. The run time of the constrained graph drawing framework and the run time of the Graphviz were compared side by side. The evaluation criteria for effectiveness measurement using the visualization approach depends on the rendering engine. The current implementation of the constrained graph drawing framework rendering engine is still in an early state and cannot be used to compare exactly against the rendering engine of the Graphviz. Thus, a set of measurements were defined to evaluate the effectiveness of layouts generated by the constrained graph drawing framework. Due to the limitations of this dissertation's scope, the implementation of our graph visualization component did not employ the horizontal coordinate assignment algorithm, Bezier curves, or polyline drawing for multilayer edges. The rendering engine simply rendered the graph layout using the vertices' positions calculated from the crossing reduction step that are stored in the relational database. The multilayer edges were drawn as a straight line. As a result, the straight line multilayer edges generated by our constrained graph drawing framework created pseudo-edge crossings that can be removed by drawing a multilayer edge as a curve or a polyline that would go around other edges. On the other hand, the Graphviz application employs a horizontal coordinate assignment and also supports drawing multilayer edges as a curve. Another limitation of the graph visualization component is that it does not restore the original direction of any cycled edges whose directions were reversed during the cycle removal algorithm. As a result, the cycle edges were displayed like other regular edges without reversing their direction back to the original direction, as was done by Graphviz. A consequence of this is that the directions of cycle edges in the generated graph layout produced by the constrained graph drawing framework are opposite to those of the cycle edges of the generated graph layout produced by Graphviz and DynaDAG. Thus, layouts generated by the constrained graph drawing framework do not look as pleasing as the layout generated by Graphviz.

To compensate for these limitations, and to have a mechanism that measures the effectiveness of the constrained graph drawing framework against Graphviz, visual evaluation criteria were defined and were used by visualization tests. To compare two layouts generated by the same dataset, the following parameters were used: (1) the number of vertices on a layer, (2) the number of layers on the layout, (3) the number of edge crossings, and (4) the similarity of the overall graph layout flow, as shown in Table 17.

Table 17. The criteria to measure effectiveness of the developed constrained framework.

Criterion	Purpose
The number of vertices on a layer	Check layer assignment step
the number layers on the layout	Check layer assignment step
the number of edge crossings	Check crossing reduction step
the similarity of the overall graph layout flow	Check the flow of the graph layout

The first visualization test set was performed to compare the layout generated by the constrained graph drawing framework against the same layout generated by Graphviz. The goal of this first test set was to validate the effectiveness of the constrained graph drawing framework in drawing static graph layouts. Five datasets were used in the first visualization test set. Two datasets were selected randomly from the Graphviz repository. The first dataset has 8 vertices but has strongly connected edges, with a total of 14 edges. The second dataset has 9 vertices and 16 edges. These first two datasets were used to measure the effectiveness of the constrained graph drawing framework in drawing graph layouts that have strongly connected edges and have many cycles. In addition to those two datasets, three datasets were randomly selected from a pool of more than 50 datasets. These datasets are subgraphs of the real-world organization and enterprise processes dataset. The third dataset has 11 vertices and 11 edges. The fourth dataset has 16 vertices and 16 edges, and the fifth dataset has 23 vertices and 23 edges. These three datasets represent typical hierarchical graph layouts, and were used to measure the effectiveness of the constrained graph drawing framework in drawing good hierarchical graph layouts. Table 18 summarizes these five datasets. For each dataset, two layouts were generated. The first layout was generated by the constrained graph drawing framework and the second layout was generated by the Graphviz application. The results of the generated graph layouts were save into images. These images were then displayed side by side for comparison.

Table 18. Dataset for the first visualization test.

Dataset name	Size	Description
Third.dot	8 vertices	Has strong connected edges
Second.dot	9 vertices	Has strong connected edges
org-parent-child-conversion-11.dot	11 vertices	Hierarchical relationship
process-parent-child-conversion-16_0.dot	16 vertices	Hierarchical relationship
org-parent-child-conversion-23_0.dot	23 vertices	Hierarchical relationship

In the same manner, to measure the elegance of the constrained graph drawing framework, which requires that the generated graph layout satisfies dynamic aesthetic criteria such as preserving layout stability, but also improves readability of the layouts after dynamic operations, seven visualization tests were conducted to cover all six dynamic operations and a static graph layout. For each test, a graph layout generated by the constrained graph drawing framework due to a dynamic operation was compared with graph layouts generated by the DynaDAG system. The two tests that cover the adding and removing of ordered constraints of vertices were not compared with the results from DynaDAG, because the current version of DynaDAG does not support ordered constraints Thus, the results of these two tests were visualized to ensure that none of the ordered constraint of vertices were violated after the graph layout was updated by a dynamic operation, and none of the vertices would be placed between the two vertices that created the order constraint.

To perform the seven aforementioned tests, a sample was randomly selected from a pool of 50 datasets. This sample was a subset of the organizational dataset and had 13 vertices and 12 edges. The first test was performed to measure the elegance of the constrained graph drawing framework in drawing a standard static graph layout. The second test was conducted to demonstrate the ability of the constrained graph drawing framework to support an added vertex and a set of edges operation. In the same manner, the remaining tests were performed to measure the ability of the constrained graph drawing framework to support adding edges, removing vertices, removing edges, and adding and removing ordered constraints.

Prior to conducting the performance and scalability tests, a preliminary test was also performed to check the design of the constrained graph drawing framework. Two large datasets were used in this preliminary test. The first dataset represented an enterprise process graph layout with 3,222 vertices. This dataset represented a real-world hierarchical graph layout with very few multilayer edges. In the same manner, the second dataset represented a real-world organizational hierarchical relationship graph with 10,370 vertices and no multilayer edges. Thus, the expectation from the test result was that the generated graph layout should not have many dummy vertices and edges. The two datasets used in the performance preliminary test are shown in Table 19. The results of these two tests were then displayed in tabular formats as shown below.

Graph layout name	Size
Number of vertices	value
Number of vertices after adding dummy vertices	value

Action	Run time (milliseconds)	Run time (milliseconds)
Time it takes to remove cycles	value	value
Time it takes to assign vertices to layers and add dummy vertices	value	value
Time it takes to reduce edge crossings	value	value
Total time it takes for the Sugiyama algorithm	value	value
Time it takes to save into database	value	value
Time it takes to render the graph	value	value

Table 19. The two datasets used in the performance preliminary test.

Dataset name	Size	Description
process-parent-child-conversion.dot	3222 vertices	Process graph
org-parent-child-conversion.dot	10370 vertices	Organizational graph

To measure the performance of the constrained graph drawing framework against the performance of Graphviz, five sets of performance tests were conducted. Table 20 shows the five performance test set names and descriptions.

The first test set was to compare the performance of the implementation of the Sugiyama heuristic against that of the algorithms used by the Graphviz application. The goal of this test set was to compare the performance of the implementation of the Sugiyama heuristic of the constrained graph drawing framework against the algorithms used by the Graphviz application. The setup for this test set comprised two test programs. The first test program, written in Java, loads the sample dataset into memory, and then executes the action procedure which in turn computes the Sugiyama heuristic run time.

The second test program, also written in Java, invokes the DOT program, which actually performs the Sugiyama heuristic, from within Java and computes the time it takes to complete the execution of the DOT program. The second test set was run to measure the performance of combining the run time of the Sugiyama heuristic and the time it takes to save the graph data into the database. Because Graphviz does not store data into a secondary storage as does the constrained graph drawing framework, the goal of the second test set was to see how the overhead incurred by the step that saves the data into the relational database impacts the overall performance of the constrained graph drawing framework.

The third performance test set was run to measure the I/O cost of retrieving data from the database back to memory for rendering graph layouts in the graph visualization component. The purpose of this set was to show the overhead of retrieving data from the database and the impact on rendering performance of using the database for storing a graph layout.

The fourth performance test set was run to measure the total run time of the graph visualization component. The purpose of this set was to validate the performance improvement of displaying the graph layout by decoupling the graph visualization from the graph editing module.

The fifth performance test set was run to compare the total run time, which basically was the combined run time of the second and third test sets, of the constrained graph drawing framework with the performance of Graphviz. The goal of this set was to show the overall performance of the constrained graph drawing framework. To compute the run time of the constrained graph drawing framework, an automatic test was written that first launched the applet. Next, for each dataset in the DOT format file, the program read data into memory and executed the Sugiyama heuristic. The program then saved the graph layout and layout snapshot into the database. Finally, the program loaded the snapshot from the database and displayed the result in the applet. The time difference between start and finish of the program was then computed as total run time. To compute the run time of Graphviz we employed the DOT command line program to test the run time of Graphviz for the majority of the dataset. Except for large datasets whose sizes were 3,000 vertices or more, a stop watch was used to compute the run time of Graphviz; this provided more accuracy than using the DOT command line program, which does not display the graph layout on the computer screen but rather executes the Sugiyama heuristic in memory, with options to save the result into a data file either in DOT format or as an image. The source code of the aforementioned tests is shown in Appendix B. In additional to these performance tests, the graph layout generated from these performance results was also used to measure the aesthetic criteria of the constrained graph drawing framework. As the number of edge crossings represents graph layout quality, a comparison of the number of crossings is included along with performance test results. The number of edge crossings of the layouts generated by the constrained graph drawing framework were compared against the same graph layout generated by Graphviz. The comparisons are presented in a tabular format.

Table 20. Five performance test sets.

Test set name	Description
Test only Sugiyama run time	Compare performance of Sugiyama algorithms implemented by constrained vs. Graphviz
Test run time of Sugiyama and database saving	Test the overhead of saving data into database
Test I/O cost of retrieving data from the database	Test the overhead of retrieving data from the database
Test run time of rendering graph layout	Test run time of render graph layout by constrained framework
Test run time of the overall action	Compare the overall performance of constrained vs. Graphviz

To test the scalability of the constrained graph layout, the four aforementioned performance test sets were conducted on three sets of data that had different sizes. The average run time results from each dataset were then plotted in a chart to show the scalability of the constrained graph drawing framework. As a result, a total of 12 sets of tests were conducted to measure both performance and scalability of the constrained graph drawing framework. Five performance test sets were run for each dataset. The first dataset, which represented a small dataset, comprised 14 samples, as shown in Table 21. Each sample had from 200 to 500 vertices. The second dataset, which represented a medium-size dataset, comprised 5 samples, as shown in Table 22. Each had from 500 to 1,000 vertices. The third dataset, which represented a large dataset, comprised 9 samples, as shown in Table 23. Each had from 1,000 to 5,000 vertices. The sizes of these samples were arbitrarily selected. The tests were set up to show the run-time results from the constrained graph layout, Graphviz, and DynaDAG. The results were then stored in a table as shown below.

Graph name	Size	Constrained	Graphviz
Name of the graph	Size of graph	Run time in milliseconds	Run time in milliseconds
……	……	……	……

Table 21. Small datasets.

Dataset name	Size (number of vertices)
org-parent-child-conversion-263	263
org-parent-child-conversion-265	265
org-parent-child-conversion-276	276
org-parent-child-conversion-277	277
org-parent-child-conversion-306	306
org-parent-child-conversion-309	309
process-parent-child-conversion-332	332
process-parent-child-conversion-337	337
org-parent-child-conversion-351	351
process-parent-child-conversion-357	357
org-parent-child-conversion-361	361
org-parent-child-conversion-386	386
process-parent-child-conversion-422	422
org-parent-child-conversion-460	460

Table 22. Medium-size datasets.

Dataset name	Size (number of vertices)
process-parent-child-conversion-526	526
org-parent-child-conversion-735	735
org-parent-child-conversion-807	807
org-parent-child-conversion-856	856
org-parent-child-conversion-888	888

Table 23. Large datasets.

Dataset name	Size (number of vertices)
org-parent-child-conversion-1063	1,063
process-parent-child-conversion-1112	1,112
org-parent-child-conversion-1444	1,444
org-parent-child-conversion-1733	1,733
process-parent-child-conversion-1849	1,849
org-parent-child-conversion-4164	4,164
process-parent-child-conversion-4495	4,495

In addition to those datasets, eight sets of real-world graph data from the Stanford large-graph dataset repository were also tested to demonstrate the scalability and shareability of the constrained graph drawing framework. Table 24 shows the eight CAIDA AS (autonomous systems) datasets from the Stanford large-graph repository that were used in the scalability test.

Table 24. Real-world graph datasets from the Stanford large-graph repository.

Dataset name	Size (number of vertices)
as-caida20040105	16,301
as-caida20040202	16,493
as-caida20040301	16,655
as-caida20040405	16,874
as-caida20040503	17,160
as-caida20040607	17,306
as-caida20040705	17,509
as-caida20040802	17,655

All the aforementioned tests were run to test the performance of the constrained graph drawing framework in loading static graph data. To measure the performance of the constrained graph drawing framework due to dynamic operations, all three small, medium, and large datasets that were used in the tests that measured the performance of the constrained graph drawing framework in loading static graph data were utilized again in these tests. The dynamic operation test scripts are in Appendix B. Table 25 shows six dynamic operations that were executed. The run time for each operation was measured. The results were then displayed in a tabular format as follows:

Graph Name	Size	Operation name	Run time
Name of the graph	Size of graph	Name of the operation	Run time in milliseconds
……	……	……	……

Table 25. Six dynamic operations.

Operation name	Purpose
Add vertex	Test adding vertex and a set of edges to the layout
Add edges	Test adding edges to the layout
Remove vertex	Test removing a vertex and all its incident edges from the layout
Remove edges	Test removing edges from the layout
Set ordered constraint	Test adding an ordered constraint to the layout
Drop ordered constraint	Test removing an ordered constraint from the layout

Resources Used

The following resources were used in this research:

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

- Hardware: Tests were performed on a Dell Latitude M6300, 2.4 GHz, 4 GB of RAM, and the OS is Windows XP SP2.
- Graphviz and DynaDAG were installed on Windows XP SP2.
- Participant: The author was the only researcher for this project.

Summary

This chapter first discussed aesthetic criteria for drawing hierarchical graph layouts and additional aesthetic criteria for incremental graph layouts. Next we reviewed 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) was then reviewed in detail, as it was a foundation for the constrained hierarchical graph drawing framework. The constrained crossing reduction problem was then reviewed, as we extended the work of Forster (2004) to solve that problem.

We then presented an abstract optimization model based on dynamic aesthetic criteria, and then translated an abstract optimization problem to a concrete optimization problem for hierarchical graph layout. The optimization problem was then incorporated into appropriate steps in the Sugiyama heuristic. We discussed the measurable goals for testing and evaluating the constrained graph drawing framework. We then described three tests that were conducted to measure the effectiveness, efficiency, and elegance of the constrained graph drawing framework.

Chapter 4

Results

This chapter presents the test results and run-time analysis of the constrained graph drawing framework. Three sets of tests were performed as described in the Testing and Evaluation section in Chapter 3. The results are presented in three sections:

- Visualization Test 1 Results, which describes the effectiveness of the constrained graph drawing framework in drawing static graph layouts
- Visualization Test 2 Results, which describes the effectiveness of the constrained graph drawing framework in drawing dynamic graph layouts
- Performance Test Results, which describes the efficiency of the constrained graph drawing framework in drawing static graph layouts and dynamic graph layouts
- Asymptotic analysis for Sugiyama algorithms and I/O cost due to database operations

Visualization Test 1 Results

The third.dot dataset used in this test was downloaded from the Graphviz web site (http://www.graphviz.org/). The data file is DOT format and is available in Appendix B. Figure 146 shows the graph layout generated by the constrained graph drawing framework, and Figure 247 shows the graph layout generated by Graphviz. The result shows that the two generated layouts were very different. The first image generated by the constrained graph drawing framework shows that the layout started at vertices 6 and 2 but the second image generated by Graphviz shows that the layout started at vertex 0. The difference indicates that one of the algorithms used by the constrained graph drawing framework could be different from the algorithm used by Graphviz.

The analysis of the third.dot data file showed that the graph layout had strong connected edges. In other words the given graph has no source nor sink, because edges are connected. As described in Chapter 3, the Greedy topological sort algorithm first finds all the sources and adds those sources into the beginning of the list. If none is found, the algorithm then finds the vertex with largest difference between outdegree and indegree and inserts that into the sorted list. In this particular dataset, vertices 6 and 2 had the largest difference between outdegree and indegree, which was two. Other vertices had either 1, 0 or negative differences. Thus, the Greedy sort algorithm inserted vertices 6 and 2 into the sorted list before all other vertices, as shown in Figure 46.

On the other hand, Graphviz uses a different sorting algorithm, which was not specified in the Graphviz user guide, but the result generated by Graphviz showed the flow of the layout nicely according to the nature of the graph layout, as shown in Figure 47. To make our test results more compatible with the layouts generated by Graphviz, we implemented a variant version of the DFS sorting algorithm, whose pseudocode was described in Chapter 3, and replaced the Greedy sorting algorithm with the modified DFS sorting algorithm. The layout generated by the constrained graph drawing framework after using the modified DFS sorting algorithm is shown in Figure 48. With the exclusion of the Bezier curve and direction of the cycled edges, the layout produced by the constrained graph drawing framework in Figure 48 had an identical overall layout flow, the same number of layers, the same number of vertices on each layer, and the same number of edge crossings, which is none in this test, as the layout produced by the Graphviz application (Figure 47).

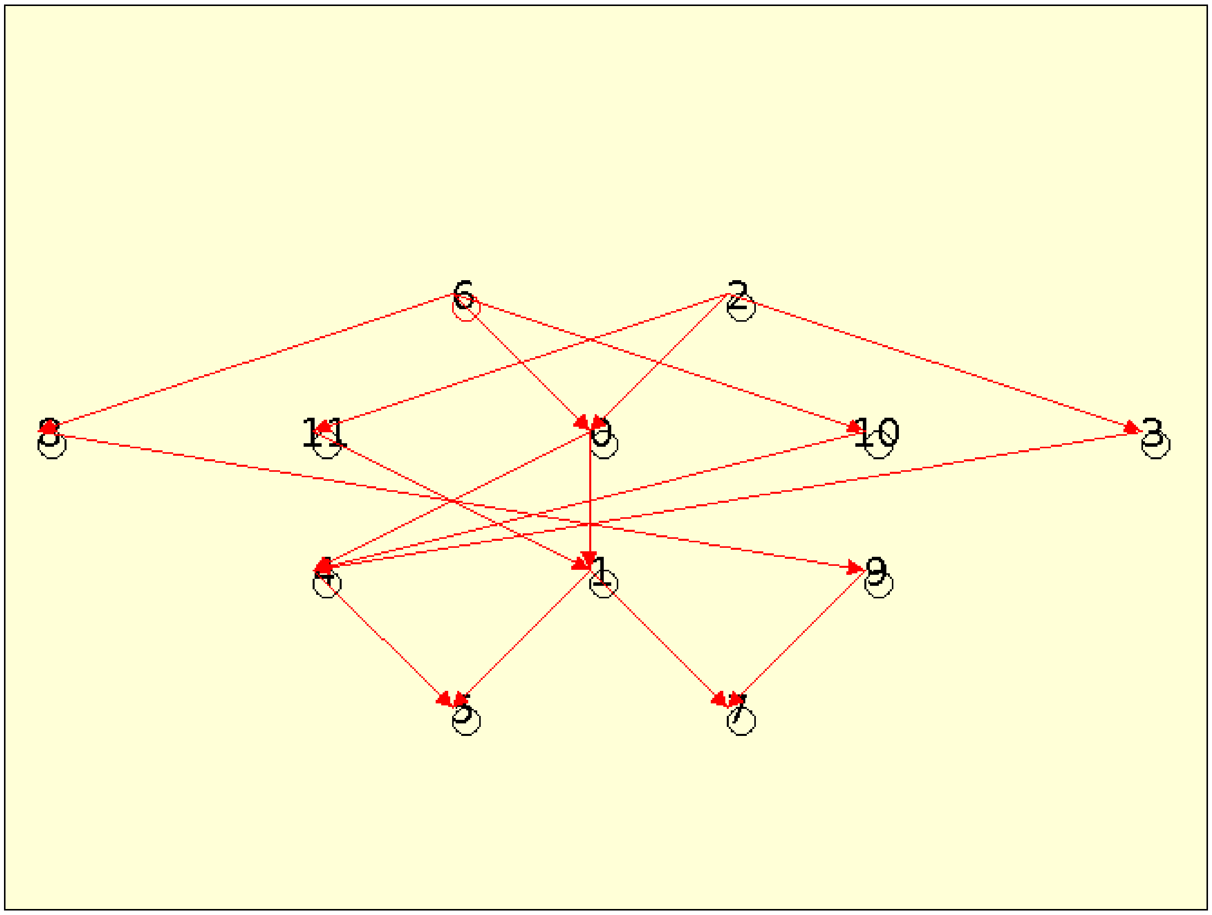


Figure 46. Layout generated by the constrained graph drawing framework.

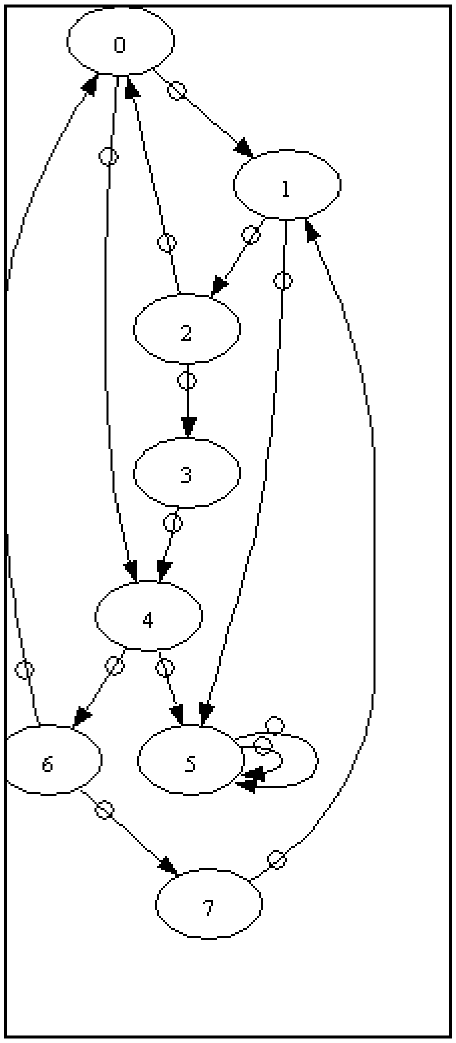


Figure 47. Layout generated by constrained Graphviz

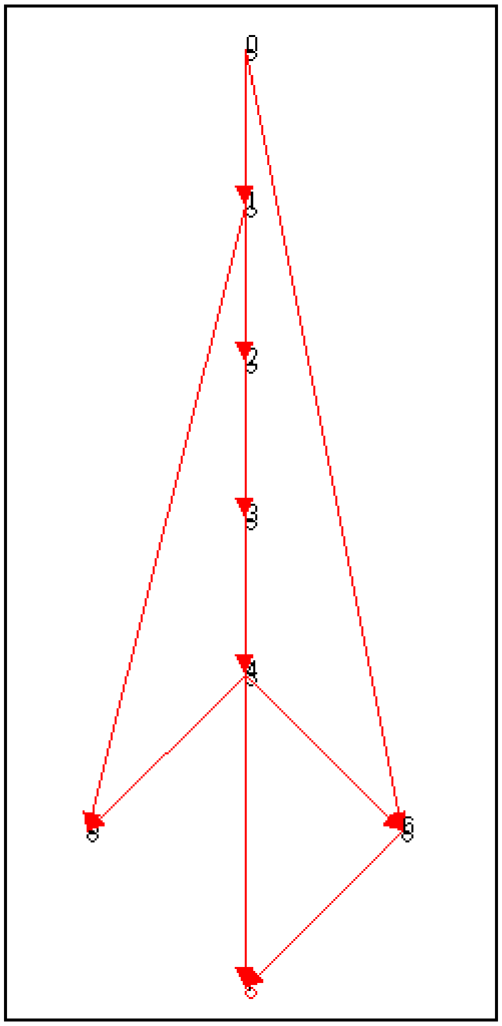


Figure 48. Layout generated after replacing the Greedy sorting algorithm with a modified DFS sorting algorithm.

The second subtest of the first visualization test was a graph layout generated from a data set called second.dot, which was also downloaded from the Graphviz repository. Figure 49 shows a graph layout generated by the constrained graph drawing framework, and Figure 50 shows the graph layout generated by Graphviz. Based on the visualization comparison specification discussed in Chapter 3, the result shows that the layout produced by the constrained graph drawing framework in Figure 49 is very similar to the layout produced by the Graphviz application (Figure 50) in terms of the graph layout flow, the number of vertices on each layer, the number of layers, and the number of edge crossings.

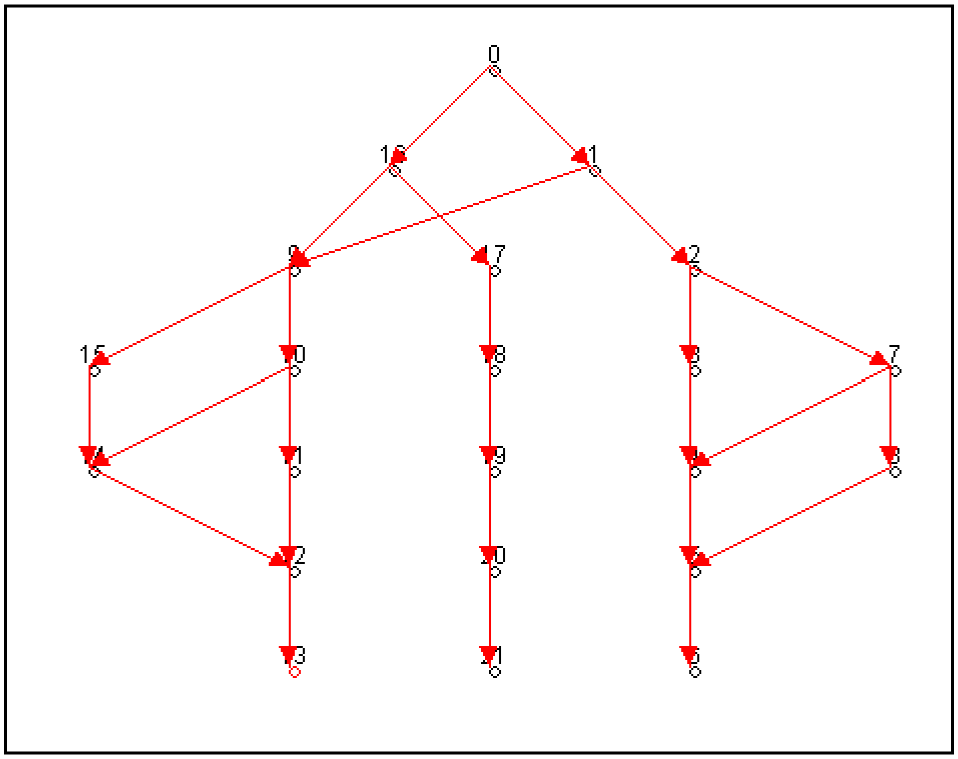


Figure 49. Layout produced by the constrained graph drawing framework

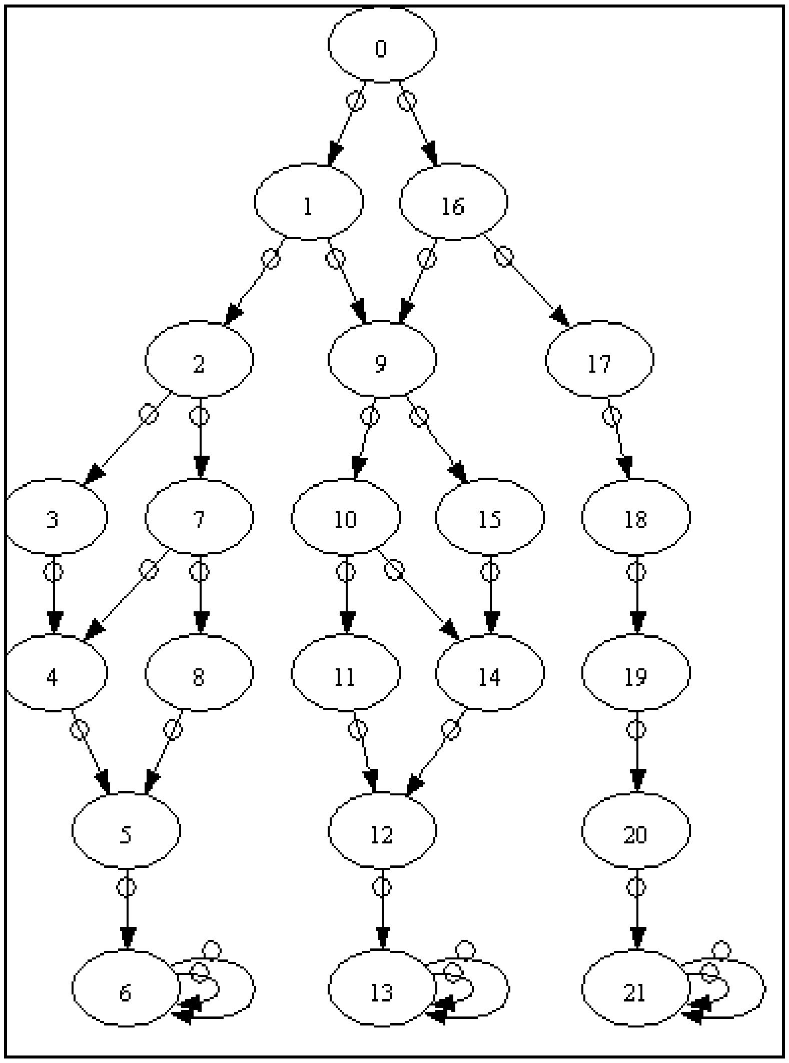


Figure 50. Layout produced by Graphviz.

While the first two tests demonstrated the effectiveness of the constrained graph drawing framework in drawing generic graph layouts, the following four tests demonstrated the effectiveness of the constrained graph drawing framework in drawing real-world hierarchical graph layouts. The third dataset was randomly selected from a pool of 56 datasets. Figure 51 shows a graph layout generated by the constrained graph drawing framework, and Figure 52 shows a graph layout generated by Graphviz. Once again, based on the visualization comparison specification discussed in Chapter 3, the result shows that the layout produced by the constrained graph drawing framework in Figure 51 is very similar to the layout in Figure 52 producing by the Graphviz application.

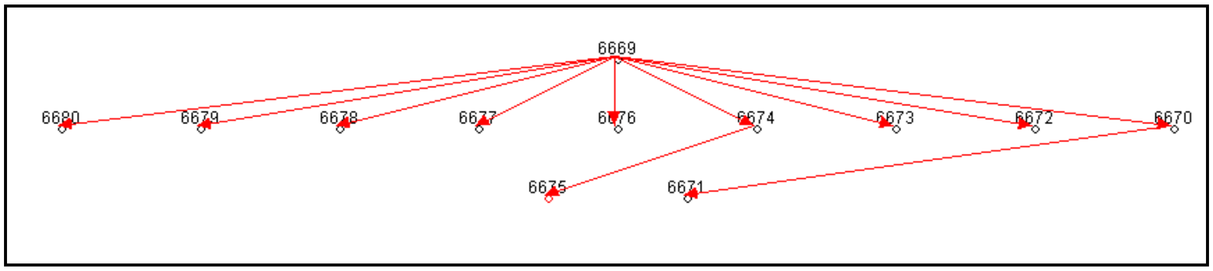


Figure 51. Layout produced by the constrained graph drawing framework

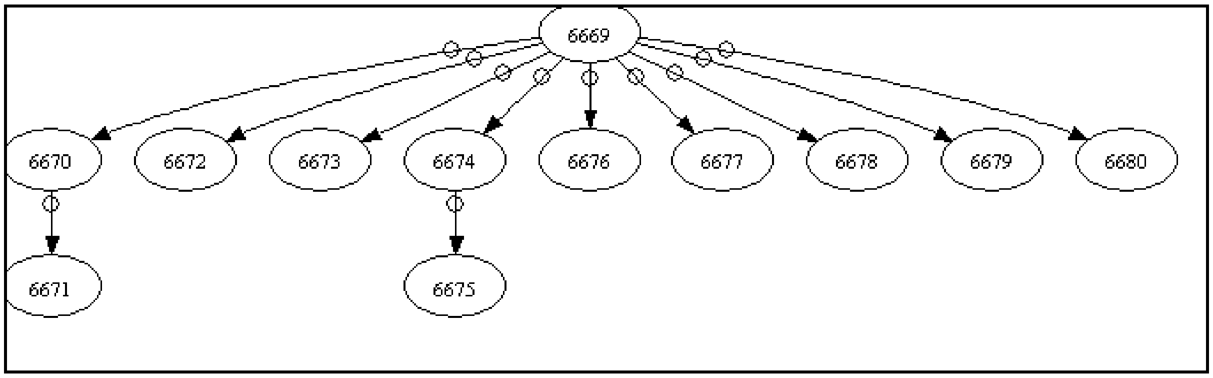


Figure 52. Layout produced by Graphviz

The fourth dataset was also randomly selected from a pool of 56 datasets. Figure 53 shows a graph layout generated by the constrained graph drawing framework, and Figure 54 shows a graph layout generated by Graphviz. The result shows that the layout produced by the constrained graph drawing framework in Figure 53 is very similar to the layout in Figure 54 produced by the Graphviz application.

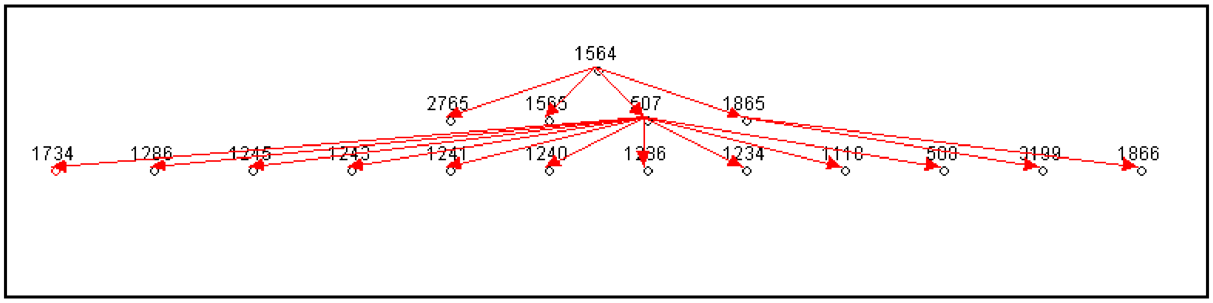


Figure 53. Layout produced by the constrained graph drawing framework

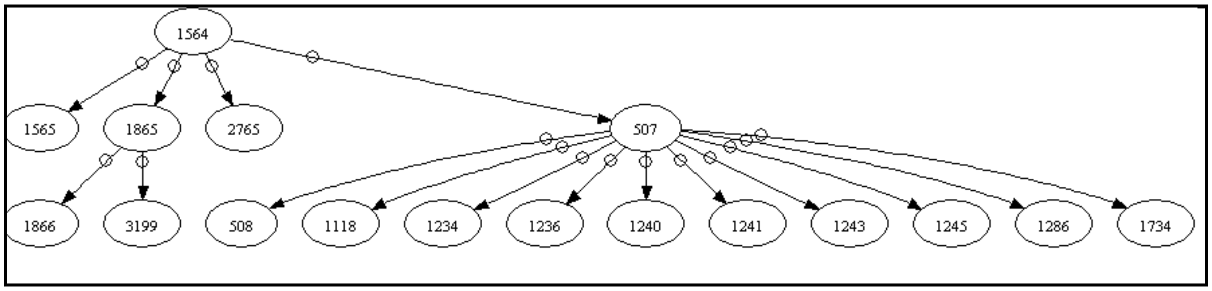


Figure 54. Layout produced by Graphviz

The fifth dataset was also randomly selected from a pool of 56 datasets. Figure 55 shows a graph layout generated by the constrained graph drawing framework, and Figure 56 shows a graph layout generated by Graphviz. The result shows that the layout produced by the constrained graph drawing framework in Figure 55 is very similar to the layout in Figure 56 produced by the Graphviz application.

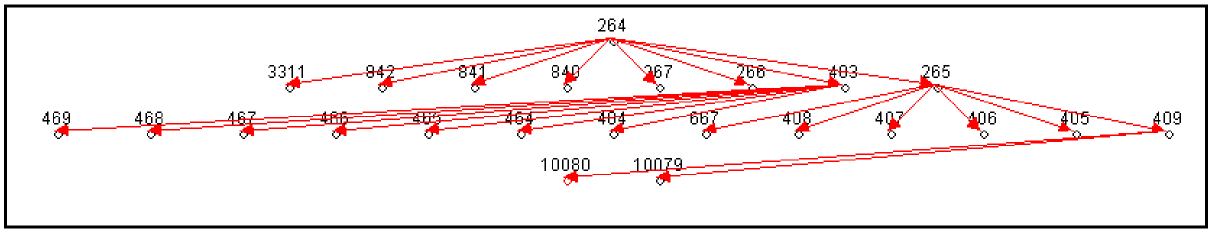


Figure 55. Layout produced by the constrained graph drawing framework

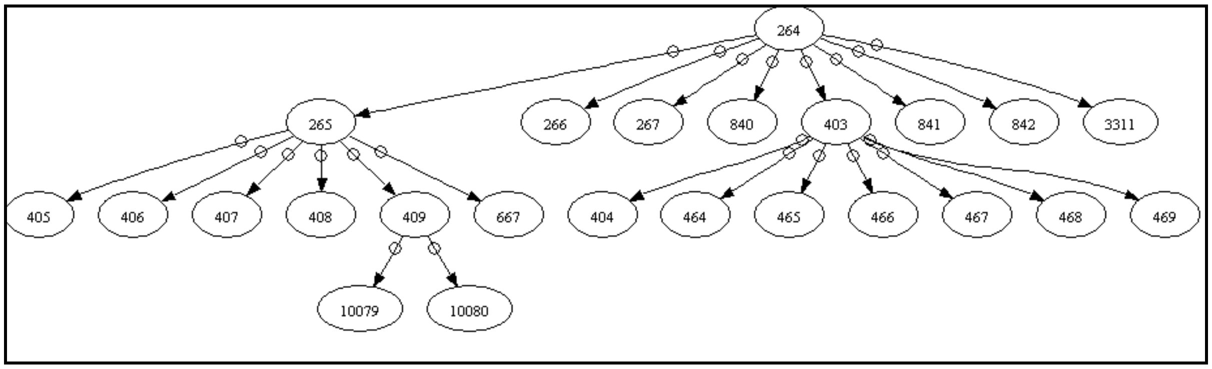


Figure 56. Layout produced by Graphviz.

Visualization Test 2 Results

This test was performed to demonstrate the elegance of the constrained graph drawing framework by preserving the layout stability and readability of the layouts due to dynamic operations. Seven scenarios were tested to cover all six of the dynamic operations and an initial static graph layout. The first test showed the first version of the graph layout that was loaded from the data file. Figures 57 and 58 show the generated layouts created by the constrained graph drawing framework and by DynaDAG respectively. The test result shows both generated layouts look very similar.

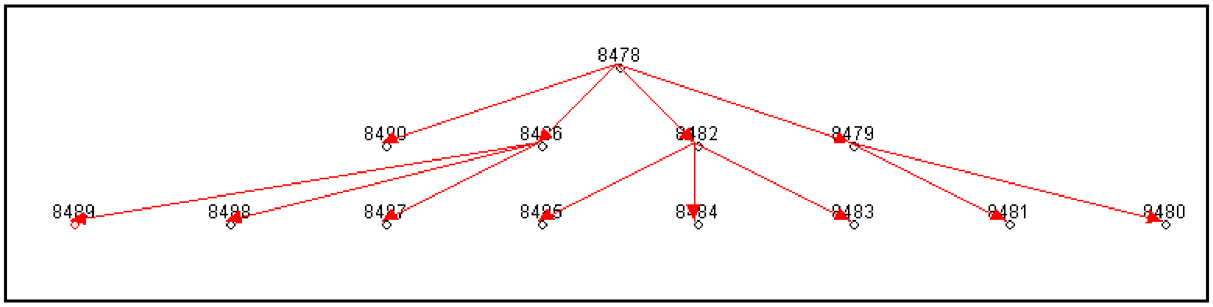


Figure 57. Layout produced by the constrained graph drawing framework

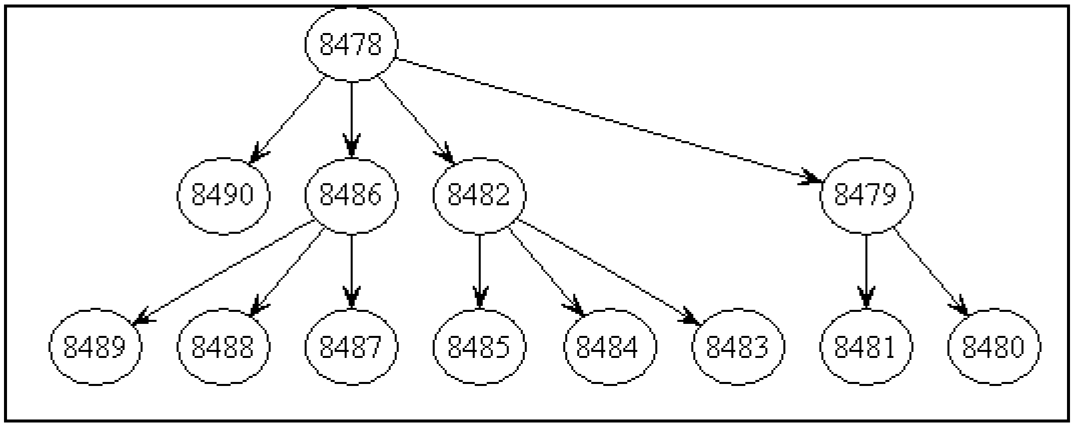


Figure 58. Layout produced by DynaDAG

The second test showed the layout after a new vertex called *1* was added. Figures 59 and 60 show the generated layouts created by the constrained graph drawing framework and by DynaDAG respectively. The similarity between the two layouts demonstrated that the constrained graph drawing framework supports the *add vertex* operation and also preserves graph layout stability by not changing the global layout after a new vertex is added.

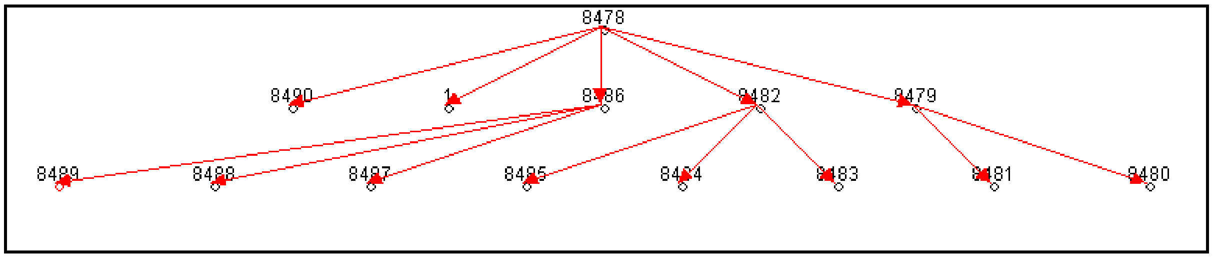


Figure 59. Layout produced by the constrained graph drawing framework

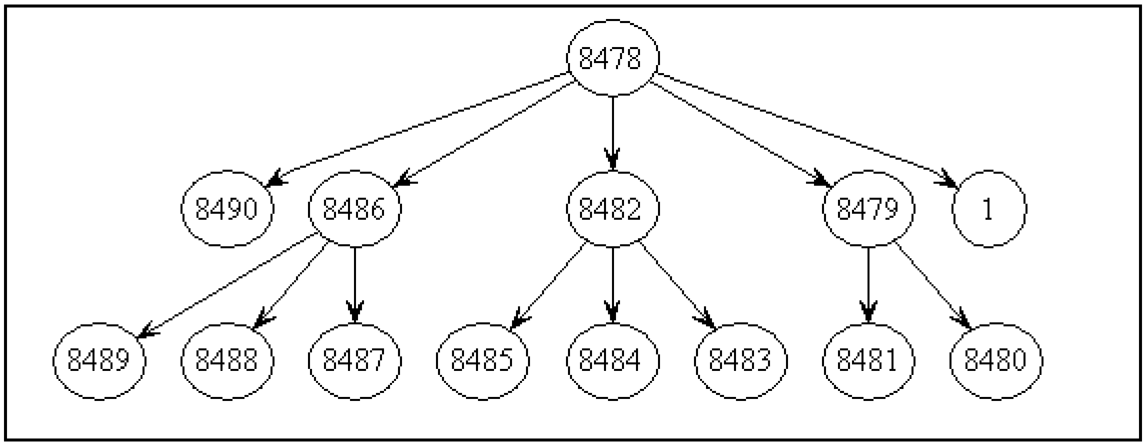


Figure 60. Layout produced by DynaDAG

The third test showed the layout after a vertex called *2* and a multilayer edge were added to the graph at layer 3. Figures 61 and 62 show the generated layouts created by the constrained graph drawing framework and by DynaDAG respectively. The result shows that there was a slight difference between the graph layout produced by the constrained drawing framework and the layout produced by DynaDAG The layout that was created by the constrained graph drawing framework had the newly created vertex 2 on layer 3, and a new multi-edge was also created that connects vertex 8478 to the newly created vertex 2. On the other hand, the layout that was created by DynaDAG had the newly created vertex 2 at layer 2 instead of layer 3. Experimenting with DynaDAG showed that the system does not allow the end-user to specify the layer specifically when adding a vertex to the existing graph layout. DynaDAG computes the new vertex's layer based on the adjacent vertex through the connected edge. In this case, the newly created vertex 2 was connected to vertex 8478, which was on layer 1. Thus, DynaDAG automatically assigned a layer 2 to vertex 2 without an option that enabled end-users to select on which layer the newly created vertex would be. The constrained graph drawing framework was designed to be flexible and enable end-users to choose which layer on which a new vertex resides. Figure 61 shows that vertex 2 was assigned to layer 3 based on user input. However, the current implementation of the constrained graph drawing framework does not automatically assign a layer to a new vertex if no layer was specified in the input. This would be an improvement of the constrained graph drawing framework.

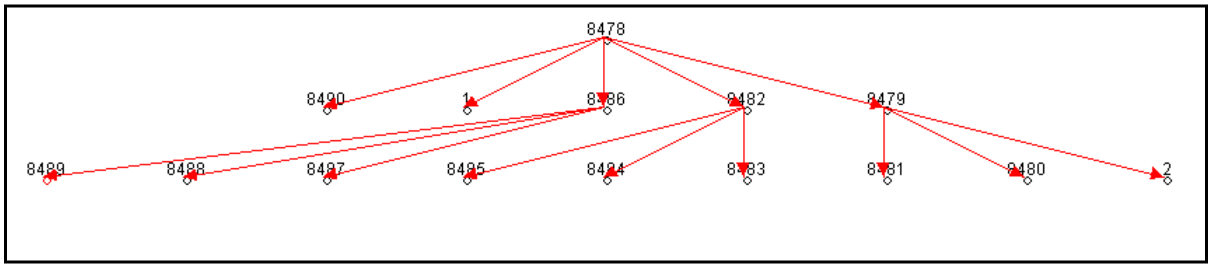


Figure 61. Layout produced by the constrained graph drawing framework

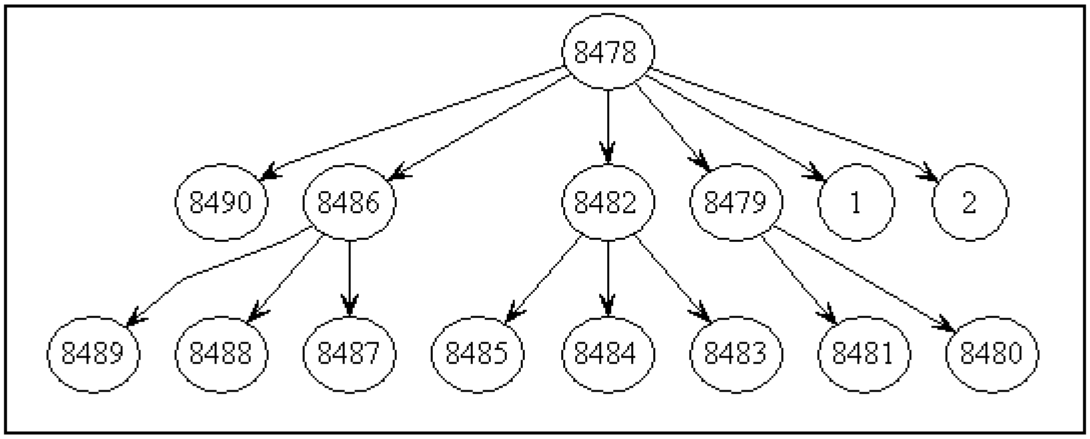


Figure 62. Layout produced by DynaDAG

The fourth test showed the layout after a simple edge was added to the graph. Figures 63 and 64 show the generated layouts created by the constrained graph drawing framework and by DynaDAG respectively. The result showed that the layout produced by the constrained drawing framework and the layout produced by DynaDAG are very similar. This test demonstrated that the constrained drawing framework supports adding edges.

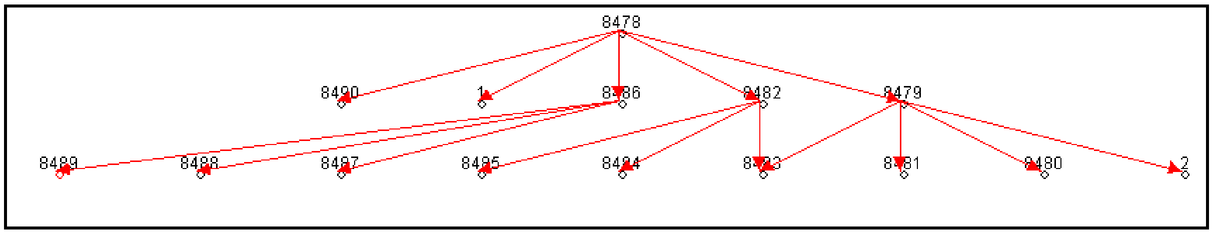


Figure 63. Layout produced by the constrained graph drawing framework

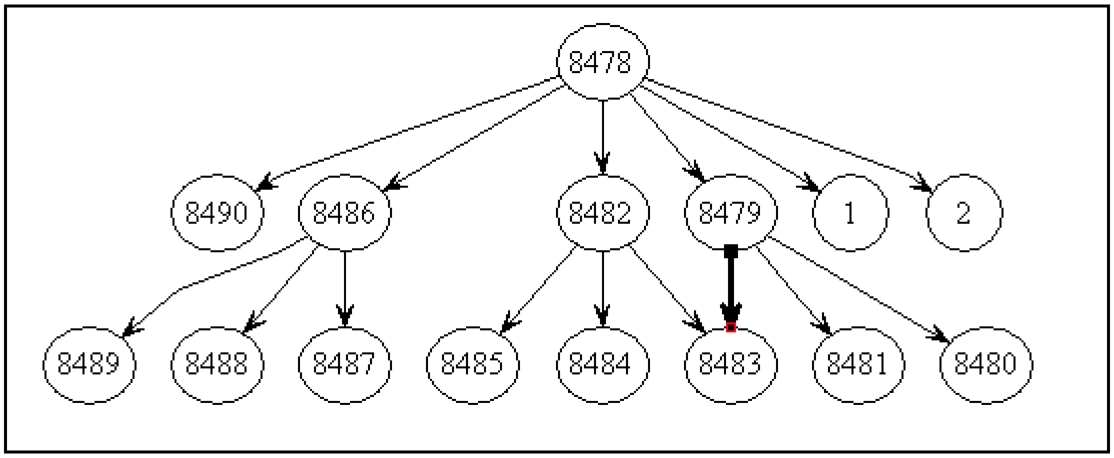


Figure 64. Layout produced by DynaDAG

The fifth test showed the layout after two ordered constraints between vertices (8482 , 8479) and (1 , 8490) were added to the graph. There is no equivalent function in DynaDAG so in this test there was no compatible layout generated by DynaDAG. Figure 65 shows the generated layout created by the constrained graph drawing framework after the ordered constraint was added. The result showed that the constrained graph drawing framework does preserve the ordered constraint of vertices while updating the graph layout.

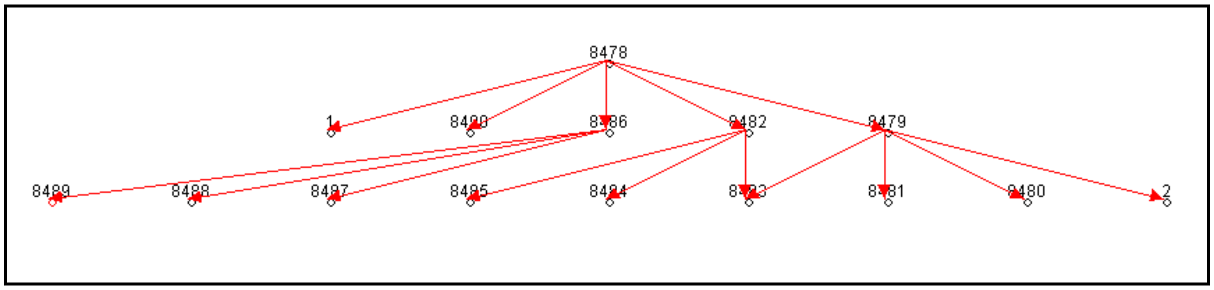


Figure 65. Layout produced by the constrained graph drawing framework

The sixth test showed the layout after a vertex and ordered constraints were removed from the graph. Figures 66 and 67 show the generated layouts created by the constrained graph drawing framework and by DynaDAG respectively. Similar to previous results, the test result showed that the layout produced by the constrained drawing framework and the layout produced by DynaDAG were very similar.

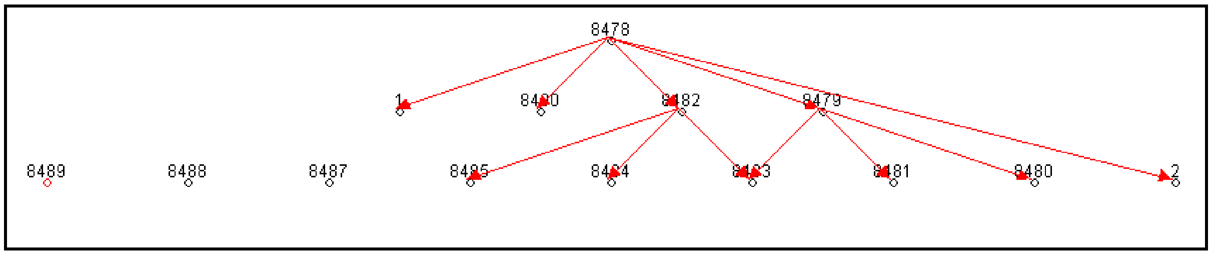


Figure 66. Layout produced by the constrained graph drawing framework

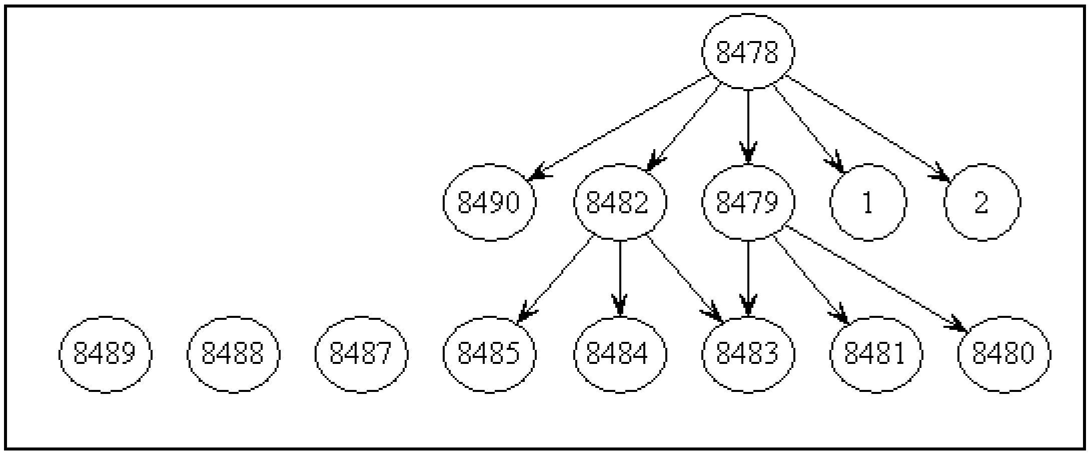


Figure 67. Layout produced by DynaDAG

The seventh test showed the layout after a number of vertices were removed from the graph. Figures 68 and 69 show the generated layouts created by the constrained graph drawing framework and by DynaDAG respectively. The test result showed that the constrained drawing framework and the DynaDAG application produced very similar layouts after several vertices had been removed.

Figure 68 shows what appear to be three edge crossings due to multilayer edge (8478, 2). Those are actually not edge crossings and can be avoided by drawing the multilayer edge (8478, 2) as a polyline going around the vertex 8479, as the Graphviz application does in drawing multilayer edges. As discussed in Chapter 3, showing pseudo-edge crossings in the layout is due to the limitations of the current implementation of the graph visualization component, which does not support Bezier curve or polyline drawing. Drawing multilayer edges as polylines would be a future enhancement for the constrained graph drawing framework. Beside this shortcoming of the design, the result showed both frameworks had a similar overall layout flow, number of vertices on each layer, number of vertices, and number of actual edge crossings.

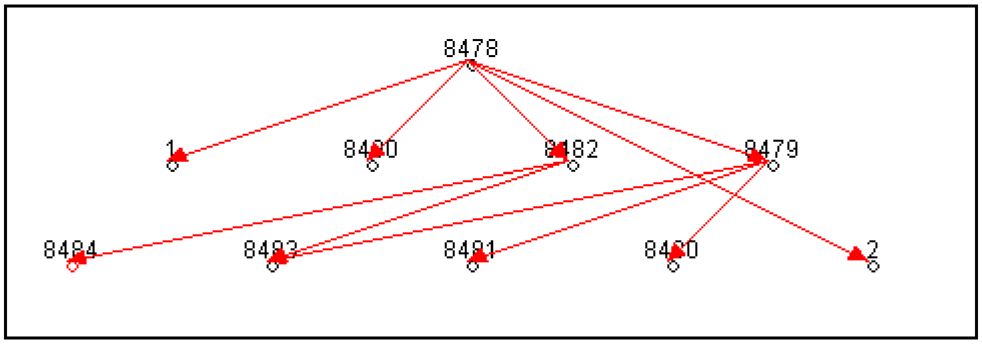


Figure 68. Layout produced by the constrained graph drawing framework

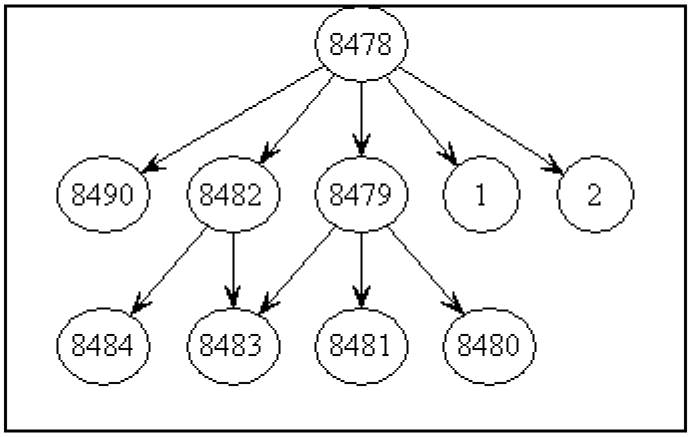


Figure 69. Layout produced by DynaDAG

Performance Test Results

The result of the preliminary performance test is shown in Table 26 and Table 27. The result showed there were three issues with the constrained graph drawing framework. The first issue is that the number of created dummy vertices was unusually high. Both datasets used for this experiment had very few or no multilayer edges. Thus, the result should not have had many dummy vertices. The second issue is that the run time of layer assignment and crossing reduction algorithms were not good. Later performance tests showed the run time of the Sugiyama heuristic was much better for Graphviz for such datasets. The third issue is that the run time of saving data into the database was not acceptable.

Table 26. Number of vertices before and after adding dummy vertices.

Graph Name	Before inserting dummy vertices	After inserting dummy vertices
Process graph	3222 vertices	18464 vertices
Organizational graph	10370 vertices	38789 vertices

Table 27. Performance result of the preliminary test.

Action	Process structure	Organizational structure
Remove cycles	0.078 s	0.758 s
Assign vertices to layers	8.073 s	57.103 s
Reduce edge crossings	5.89 s	33.912 s
Total time it takes for Sugiyama algorithm	14.041 s	91.773 s
Save into database	33.005 s	71.415 s
Render to graph (applet)	1.8 s	5.237 s

After reviewing the algorithms, we noticed we used an arbitrary value to limit the number of vertices per layer in the layer assignment algorithm. This value did impact how the layer assignment algorithm assigned vertices to layers and created dummy vertices. For example, if this value was less than the actual number of vertices on a layer due to the nature of the data structure, the layer assignment algorithm created pseudo-multilayer edges by pushing vertices down to the next layers when the number of vertices per layer was larger than the maximum allowable value per layer. Those intentional extra dummy vertices were then used to created pseudo-multilayer edges. We called those pseudo-multilayer edges because the edges would not be created if the value limiting the width of a layer was larger than the actual number of vertices on that layer. As a result, the number of created dummy vertices impacted the performance of the layer assignment and crossing reduction algorithms. To resolve the number of dummy vertices and the run time of the Sugiyama heuristic, the value that limits the width of layers in layer assignment was set to a very large number.

To attack the poor performance of the step that saves data into the database, we reviewed the program and noticed that the constrained graph drawing framework utilized the object-to-relational-mapping API called Hibernate to save data from memory into the database. This approach saves programming time but it added more overhead to the performance of the algorithm. To improve the performance of this step the program was modified to use direct relational calls instead of the object-to-relational-mapping API when saving data from memory to object-to-relational-mapping database. The preliminary test was run again after changes were made to the constrained graph drawing framework.

The results of the retest are shown in Tables 28 and 29. The result showed the number of vertices before and after adding dummy vertices was much better. To validate the number of vertices and the correctness of the design, two graph layouts were produced and compared side by side. The first layout was generated by the constrained graph drawing framework and the second layout was generated by the Graphviz program. To compensate for the poor visibility of both layouts due to their size, a SQL query was run that computed the number of layers of the layout generated by the constrained graph drawing framework. The SQL queries used in this analysis are included in Appendix D. The query result showed there were 11 layers for the process graph layout generated by the constrained graph drawing framework. For the same process graph layout generated by Graphviz the count was performed manually by looking at the zoomed-in image. The image showed the number of layers was approximately 10. Furthermore, the zoomed-out image showed that the layer width was much greater than the height, which indicates that the Graphviz algorithm did not restrict layer width based on an arbitrary value. Thus, this result showed that even though in theory limiting the width of a layer is considered to enhance the readability of a graph layout, real-world data structures showed that the width of a layer should be based on the actual number of vertices on the layer, not an arbitrary value that limits layer width for some data structures.

Table 28. Number of vertices before and after adding dummy vertices (retest).

Graph name	Before inserting dummy vertices	After inserting dummy vertices
Process graph	3,222 vertices	3,674 vertices
Organizational graph	10,370 vertices	10,370 vertices

Table 29. Performance result of the preliminary test (retest).

Action	Process structure	Organizational structure
Remove cycles	0.272 s	1.549 s
Assign vertices to layers	0.588 s	0.307 s
Reduce edge crossings	0.262 s	1.054 s
Total time it takes for Sugiyama algorithm	1.122 s	2.91 s
Save into database	2.753 s	6.113 s
Render to graph (applet)	1.309 s	2.359 s

The result of the retest also showed that the performance of the Sugiyama heuristic improved dramatically. To ensure that the overall performance of the constrained graph drawing framework is compatible with that of Graphviz and DynaDAG, we generated the same graph layouts using Graphviz and DynaDAG and computed the run time using a stop watch. The result in Table 30 shows that the overall performance of the constrained graph drawing framework is better than that of Graphviz. The reason for this is that the constrained graph drawing framework did not implement horizontal coordinate assignment. Because the difference in run time between the constrained graph drawing framework performance and the Graphviz performance is no more than 6 seconds for the process dataset and 60 seconds for the organizational dataset, we concluded that the performance of the constrained graph drawing framework is compatible with that of Graphviz if the run time of the horizontal coordinate assignment algorithm is taken into account. The comparison with DynaDAG was inconclusive because DynaDAG could not produce a graph layout for those two data structures; it hung while the test was performed. As the current version of the DynaDAG framework could not handle large datasets, Graphviz was used in our performance test for static graph layouts instead. As a result, to measure performance we compared our results against the performance of Graphviz for generating static graph layouts. For performance due to dynamic operations, tests were conducted and the results displayed in a tabular format.

Table 30. Performance comparisons between constrained, Graphviz, and DynaDAG.

Graph name	Size	Constrained	Graphviz	DynaDAG
Process data structure	3,222	**5.184 s**	**12 s**	Inconclusive
Organizational structure	10,370	**11.382 s**	**79 s**	Inconclusive

The following section presents the results of 12 performance tests, 4 for each of the 3 datasets. The run times were all computed in milliseconds and the results are displayed in tables. In each table, the first column contains the name of the graph, the second column shows the size of the dataset, the third column shows the run time of the constrained graph drawing framework, and the fourth column displays the run time of Graphviz.

The first performance tests were run to measure the run time of the Sugiyama algorithm on small datasets. The results of these tests are shown in Table 31. With the exception of the first result (second row), which was discarded due to an anomaly that could have been the time it took to initialize the Graphviz application, the test results indicate that the performance of the Sugiyama algorithms implemented by the constrained graph drawing framework is comparable with that of the algorithms employed by Graphviz. The difference in milliseconds between the constrained graph drawing framework and the performance of Graphviz is due to the fact that the constrained graph drawing framework did not include a horizontal coordinate assignment algorithm in its implementation of the Sugiyama heuristic. As a result, we concluded that the run time of the two frameworks should be closer if the horizontal coordinate assignment algorithm had been implemented.

Table 31. Results of the first performance tests on small datasets.

Graph name	Size (vertices)	Constrained (milliseconds)	Graphviz (milliseconds)
org-parent-child-conversion-263	263	31	78
org-parent-child-conversion-265	265	141	1,172
org-parent-child-conversion-276	276	47	78
org-parent-child-conversion-277	277	31	63
org-parent-child-conversion-306	306	32	62
org-parent-child-conversion-309	309	31	94
process-parent-child-conversion-332	332	16	94
process-parent-child-conversion-337	337	15	79
org-parent-child-conversion-351	351	47	78
process-parent-child-conversion-357	357	62	94
org-parent-child-conversion-361	361	47	78
org-parent-child-conversion-386	386	32	78
process-parent-child-conversion-422	422	31	78
org-parent-child-conversion-460	460	46	94

The second performance tests were run to measure the time taken to perform the Sugiyama algorithm and to save data into the relational database. The results of these tests are shown in Table 32. Though the test results show that Graphviz outperformed the constrained graph drawing framework in terms of run time due to the latter's overhead of saving data into the database, the benefit of storing graph layout snapshots into a relational database outweighed the overhead, as shown in the results of the fourth test.

Table 32. Results of the second performance tests on small datasets.

Graph name	Size (vertices)	Constrained (milliseconds)	Graphviz (milliseconds)
org-parent-child-conversion-263	263	204	62
org-parent-child-conversion-265	265	344	78
org-parent-child-conversion-276	276	219	62
org-parent-child-conversion-277	277	203	63
org-parent-child-conversion-306	306	250	78
org-parent-child-conversion-309	309	219	63
process-parent-child-conversion-332	332	219	94
process-parent-child-conversion-337	337	203	78
org-parent-child-conversion-351	351	234	78
process-parent-child-conversion-357	357	266	78
org-parent-child-conversion-361	361	234	78
org-parent-child-conversion-386	386	266	62
process-parent-child-conversion-422	422	266	94
org-parent-child-conversion-460	460	359	78

The third performance tests were run to measure the time taken to retrieve data from the relational database back into memory. The results of the data retrieval tests are shown in Table 33. The results show that the I/O cost of database retrieval is linear to the size of the dataset except for the first result, which could be from when the database connection was first established. The average time taken to load a dataset with a size of a few hundred vertices is approximately less than 100 milliseconds, which is acceptable for loading and rendering graph layouts on the Internet.

Table 33. Results of the database retrieval cost tests on small datasets.

Graph name	Size (vertices)	Constrained (milliseconds)	Graphviz (milliseconds)
org-parent-child-conversion-263	263	109	NA
org-parent-child-conversion-265	265	62	NA
org-parent-child-conversion-276	276	78	NA
org-parent-child-conversion-277	277	63	NA
org-parent-child-conversion-306	306	62	NA
org-parent-child-conversion-309	309	47	NA
process-parent-child-conversion-332	332	47	NA
process-parent-child-conversion-337	337	62	NA
org-parent-child-conversion-351	351	63	NA
process-parent-child-conversion-357	357	47	NA
org-parent-child-conversion-361	361	46	NA
org-parent-child-conversion-386	386	47	NA
process-parent-child-conversion-422	422	47	NA
org-parent-child-conversion-460	460	47	NA

The rendering performance tests were run to measure the run time of the visualization component, which combines the time taken to retrieve data from the database and the time taken to render the graph layout to the Internet. The results of these tests are shown in Table 34. The test results indicate that the performance is acceptable for rendering graph layouts on the Internet in real time even with the cost of database retrieval.

Table 34. Results of the rendering performance tests on small datasets.

Graph name	Size (vertices)	Constrained (milliseconds)	Graphviz (milliseconds)
org-parent-child-conversion-263	263	78	NA
org-parent-child-conversion-265	265	109	NA
org-parent-child-conversion-276	276	94	NA
org-parent-child-conversion-277	277	78	NA
org-parent-child-conversion-306	306	94	NA
org-parent-child-conversion-309	309	78	NA
process-parent-child-conversion-332	332	78	NA
process-parent-child-conversion-337	337	94	NA
org-parent-child-conversion-351	351	94	NA
process-parent-child-conversion-357	357	110	NA
org-parent-child-conversion-361	361	109	NA
org-parent-child-conversion-386	386	109	NA
process-parent-child-conversion-422	422	94	NA
org-parent-child-conversion-460	460	109	NA

The fourth performance tests were run to compare the total run time of the constrained graph drawing framework with that of the Graphviz application. The results of these tests are shown in Table 35. The results show that the constrained graph drawing framework outperformed Graphviz in all tests. As mentioned earlier, the test did not take into account the time it takes to compute the horizontal assignment due to the fact that the constrained graph drawing framework did not implement the horizontal assignment algorithm. Thus, the performance of the constrained graph drawing framework should be close to the performance of Graphviz in drawing graph layouts whose sizes are from 200 to 500 vertices.

Table 35. Results of the fifth performance tests on small datasets.

Graph name	Size (vertices)	Constrained (milliseconds)	Graphviz (milliseconds)
org-parent-child-conversion-263	263	250	984
org-parent-child-conversion-265	265	453	765
org-parent-child-conversion-276	276	343	813
org-parent-child-conversion-277	277	297	875
org-parent-child-conversion-306	306	297	703
org-parent-child-conversion-309	309	297	1,140
process-parent-child-conversion-332	332	297	1,578
process-parent-child-conversion-337	337	281	1,172
org-parent-child-conversion-351	351	327	1,266
process-parent-child-conversion-357	357	406	1,032
org-parent-child-conversion-361	361	328	1,407
org-parent-child-conversion-386	386	327	1,265
process-parent-child-conversion-422	422	375	2,531
org-parent-child-conversion-460	460	406	1,531

Similar to the tests that were run on small datasets, the following performance tests were run on medium-sized datasets. The first performance tests were run to measure the run time of the Sugiyama algorithm. The results shown in Table 36 indicate that the Sugiyama algorithms implemented by the constrained graph drawing framework perform well against the algorithms employed by Graphviz for datasets whose sizes are from 500 to 1,000 vertices.

Table 36. Results of the first performance tests on medium-sized datasets.

Graph name	Size (vertices)	Constrained (milliseconds)	Graphviz (milliseconds)
process-parent-child-conversion-526	526	31	125
org-parent-child-conversion-735	735	94	109
org-parent-child-conversion-807	807	125	110
org-parent-child-conversion-856	856	125	125
org-parent-child-conversion-888	888	172	141

The second performance tests were conducted to measure the time it takes to execute the Sugiyama algorithm to save data into a relational database on medium-sized datasets of from 500 to 1,000 vertices. The results of these tests are shown in Table 37. The results show that Graphviz outperformed the constrained graph drawing framework due to the latter's overhead from saving data into the database.

Table 37. Results of the second performance tests on medium-sized datasets.

Graph name	Size (vertices)	Constrained (milliseconds)	Graphviz (milliseconds)
process-parent-child-conversion-526	526	360	109
org-parent-child-conversion-735	735	531	125
org-parent-child-conversion-807	807	578	125
org-parent-child-conversion-856	856	578	125
org-parent-child-conversion-888	888	609	141

The third tests were run to measure the time it takes to load data from the database to memory of the visualization component for medium-sized datasets of from 500 to 1,000 vertices. The test results shown in Table 38 indicate that the average time taken to retrieve data from the database was strictly less than 100 milliseconds for a dataset whose size was less than 1000 vertices.

Table 38. Results of the data retrieval performance tests on medium-sized datasets.

Graph name	Size (vertices)	Constrained (milliseconds)	Graphviz (milliseconds)
process-parent-child-conversion-526	526	63	NA
org-parent-child-conversion-735	735	63	NA
org-parent-child-conversion-807	807	47	NA
org-parent-child-conversion-856	856	63	NA
org-parent-child-conversion-888	888	62	NA

A rendering test was run to measure the run time of the visualization component for medium-sized datasets of from 500 to 1,000 vertices. The test results shown in Table 39 indicate that even with the I/O overhead the overall run time was reasonable for rendering graph layouts on the Internet. Although decoupling the graph visualization component from the graph editor and saving graph layouts to the database incurred I/O overhead, this approach enabled the constrained graph drawing framework to render graph layouts over the Internet acceptably quickly and allowed a move away from traditional desktop environment to distributed environments such as client-server.

Table 39. Results of the rendering performance tests on medium-sized datasets.

Graph name	Size (vertices)	Constrained (milliseconds)	Graphviz (milliseconds)
process-parent-child-conversion-526	526	110	NA
org-parent-child-conversion-735	735	156	NA
org-parent-child-conversion-807	807	141	NA
org-parent-child-conversion-856	856	157	NA
org-parent-child-conversion-888	888	156	NA

The fifth performance tests were run to compare the total run time of the constrained graph drawing framework with the run time of the Graphviz application for medium-sized datasets. The results in Table 40 show that the performance of the constrained graph drawing framework should be close to the performance of Graphviz when drawing graph layouts sized from 500 to 1,000 vertices, if the test takes into account the run time of the horizontal coordinate assignment algorithm.

Table 40. Results of the fifth performance tests on medium-sized datasets.

Graph name	Size (vertices)	Constrained (milliseconds)	Graphviz (milliseconds)
process-parent-child-conversion-526	526	453	1,781
org-parent-child-conversion-735	735	640	625
org-parent-child-conversion-807	807	812	1,203
org-parent-child-conversion-856	856	766	1,765
org-parent-child-conversion-888	888	766	657

The following are results of performance tests run on large datasets. In the same manner as the prior tests, the first performance tests were run to measure the run time of the implementation of the Sugiyama algorithm. The second tests measured the time to perform the Sugiyama algorithm and to save data into a relational database. The third tests measured data retrieval cost. The fourth tests measured the run time of the visualization component. The fifth performance test was run to compare the total run time of the constrained graph drawing framework with the run time of the Graphviz application. The results of these tests are shown in Tables 41, 42, 43, 44, and 45 respectively. The test results show that the performance of the constrained graph drawing framework on large datasets compares well with that of Graphviz.

Table 41. Results of the first performance tests on large datasets.

Graph name	Size (vertices)	Constrained (milliseconds)	Graphviz (milliseconds)
org-parent-child-conversion-1063	1,063	172	156
process-parent-child-conversion-1112	1,112	94	281
org-parent-child-conversion-1444	1,444	250	219
org-parent-child-conversion-1733	1,733	438	234
process-parent-child-conversion-1849	1,849	234	625
org-parent-child-conversion-4164	4,164	1,844	1,531
process-parent-child-conversion-4495	4,495	860	3,375

Table 42. Results of the second performance tests on large datasets.

Graph name	Size (vertices)	Constrained (milliseconds)	Graphviz (milliseconds)
org-parent-child-conversion-1063	1,063	813	156
process-parent-child-conversion-1112	1,112	609	281
org-parent-child-conversion-1444	1,444	1,000	203
org-parent-child-conversion-1733	1,733	1,390	219
process-parent-child-conversion-1849	1,849	1,172	641
org-parent-child-conversion-4164	4,164	3,984	1,454
process-parent-child-conversion-4495	4,495	2,766	3,313

Table 43. Results of the data retrieval performance tests on large datasets.

Graph name	Size (vertices)	Constrained (milliseconds)	Graphviz (milliseconds)
org-parent-child-conversion-1063	1,063	78	NA
process-parent-child-conversion-1112	1,112	63	NA
org-parent-child-conversion-1444	1,444	141	NA
org-parent-child-conversion-1733	1,733	78	NA
process-parent-child-conversion-1849	1,849	93	NA
org-parent-child-conversion-4164	4,164	140	NA
process-parent-child-conversion-4495	4,495	141	NA

Table 44. Results of the rendering performance tests on large datasets.

Graph name	Size (vertices)	Constrained (milliseconds)	Graphviz (milliseconds)
org-parent-child-conversion-1063	1,063	203	NA
process-parent-child-conversion-1112	1,112	281	NA
org-parent-child-conversion-1444	1,444	235	NA
org-parent-child-conversion-1733	1,733	282	NA
process-parent-child-conversion-1849	1,849	328	NA
org-parent-child-conversion-4164	4,164	797	NA
process-parent-child-conversion-4495	4,495	594	NA

Table 45. Results of the fifth performance tests on large datasets.

Graph name	Size (vertices)	Constrained (milliseconds)	Graphviz (milliseconds)
org-parent-child-conversion-1063	1,063	985	2,109
process-parent-child-conversion-1112	1,112	860	3,172
org-parent-child-conversion-1444	1,444	1,235	1,812
org-parent-child-conversion-1733	1,733	1,625	1,360
process-parent-child-conversion-1849	1,849	1,687	3,063
org-parent-child-conversion-4164	4,164	4,609	10,000
process-parent-child-conversion-4495	4,495	3,500	6,922

To measure the scalability of the constrained graph drawing framework, 12 test results from 3 datasets were combined and plotted to create a performance chart, as shown in Figure 70. The detailed performance chart shows that both frameworks were scalable and could render graphs whose size was up to 4,000 vertices within 10 seconds.

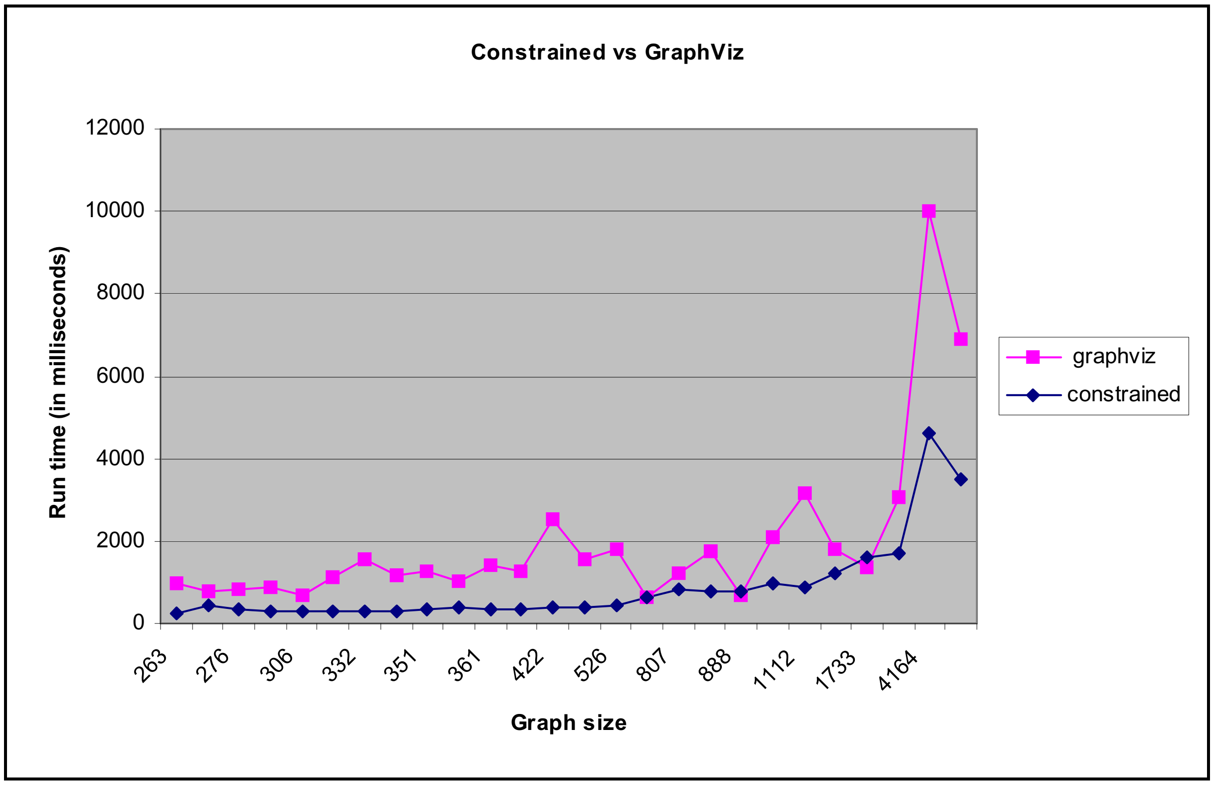


Figure 70. Performance of the constrained graph drawing framework versus Graphviz

Furthermore, these test results showed the overall run time of the two frameworks from loading data into the memory to rendering the graph layout without taking into account the decoupling of the graph visualization from the graph editing component. In real-world applications such as the enterprise process modeling tool, few modelers make changes to the model but many more viewers view the graph layouts on line. Once the model and snapshots were saved into a database, the constrained graph drawing framework not only rendered the graph layout faster than did Graphviz, but also scaled better for large graphs, as shown in Figure 71. Please note in Figure 71 the run-time comparison between the overall run time of Graphviz and the run time of the visualization component of the constrained graph drawing framework.

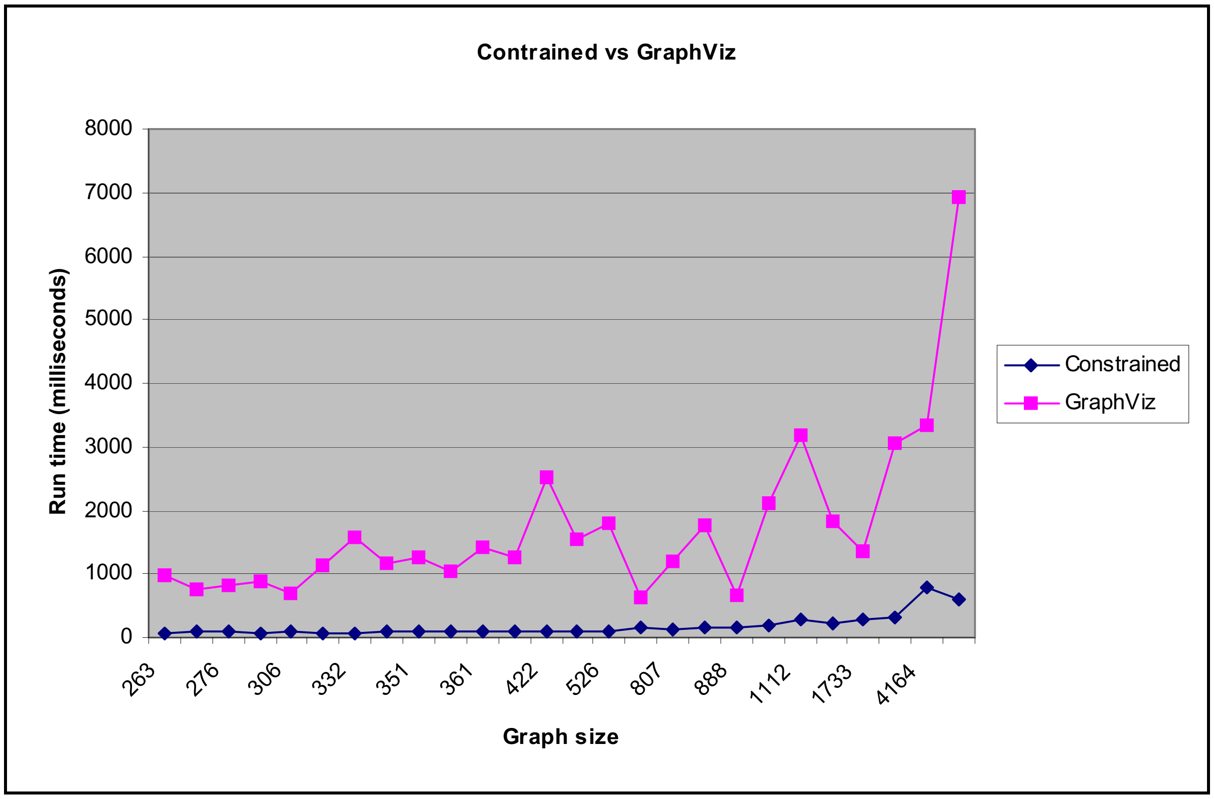


Figure 71. Rendering performance of the constrained graph drawing framework versus Graphviz

To demonstrate that the layouts generated by the constrained graph drawing framework satisfy the aesthetic criteria, especially for the number of edge crossings, the edge crossings of layouts generated by the constrained graph drawing framework in the performance tests were computed and compared against the number of crossings in the same layouts generated by Graphviz. The comparison results are presented in Table 46. The comparison shows that both frameworks produced the optimal solution, where layouts have zero edge crossings if they have no edge crossings by nature. For example, most organizational graph layouts do not have edge crossings by nature. For layouts that have edge crossings by nature, Graphviz produces layouts with fewer edge crossings, except for two large graph layouts whose sizes are 1849 and 4495. For those two layouts, the constrained graph drawing framework produces layouts with fewer edge crossings. A detailed analysis was performed on those two large graph layouts. For the graph layout whose size was 1849, the layout generated by the constrained framework had 8 layers; the layout generated by Graphviz had 10 layers. In other words, Graphviz limited the total width of the layout due to screen real state and created dummy vertices, which created more edge crossings. The constrained graph framework allows the width of a layout to be as wide as the maximum number of incident edges a vertex has. Consequently, its layout has fewer edge crossings but is wider. Similarly, for the graph whose size was 4495, the layout generated by the constrained graph framework has 11 layers with fewer edge crossings but the layout is wider. On the other hand, the layout generated by Graphviz has 13 layers with more edge crossings but is narrower. With improvements in the crossing reduction sweeping algorithm, the constrained graph drawing framework should generate layouts with fewer edge crossings if applicable.

Table 46. Number of edge crossings comparison.

Graph name	Size	Constrained (edge crossings)	Graphviz (edge crossings)
org-parent-child-conversion-263	263	0	0
org-parent-child-conversion-265	265	0	0
org-parent-child-conversion-276	276	0	0
org-parent-child-conversion-277	277	0	0
org-parent-child-conversion-306	306	0	0
org-parent-child-conversion-309	309	0	0
process-parent-child-conversion-332	332	8	1
process-parent-child-conversion-337	337	171	35
org-parent-child-conversion-351	351	0	0
process-parent-child-conversion-357	357	44	0
org-parent-child-conversion-361	361	0	0
org-parent-child-conversion-386	386	0	0
process-parent-child-conversion-422	422	66	0
org-parent-child-conversion-460	460	0	0
process-parent-child-conversion-526	526	88	1
org-parent-child-conversion-735	735	0	0
org-parent-child-conversion-807	807	0	0
org-parent-child-conversion-856	856	0	0
org-parent-child-conversion-888	888	0	0
org-parent-child-conversion-1063	1,063	0	0
process-parent-child-conversion-1112	1,112	815	259
org-parent-child-conversion-1444	1,444	0	0
org-parent-child-conversion-1733	1,733	0	0
process-parent-child-conversion-1849	1,849	2,846	3,392
org-parent-child-conversion-4164	4,164	0	0
process-parent-child-conversion-4495	4,495	26,858	31,369

To demonstrate the scalability of the constrained graph layout framework for real-world graph layouts, eight datasets of CAIDA AS (autonomous systems) from the Stanford large-graph data repository were tested. Each dataset was loaded into Graphviz, DynaDAG, and the constrained graph drawing framework respectively. None of the frameworks was able to render the graph layouts even after 2 hours of running and consuming almost 100 % of computer resources. The tests were terminated after 2 hours of running. To test the scalability and shareability of the constrained graph drawing framework, another test was performed. First, the size of the graph data was reduced to approximately 4000 vertices by removing the vertices and edges. The deleted vertices and edges were saved so they could later be added back to the graph through dynamic operations. After loading and saving the initial graph layouts, the constrained graph drawing framework added all the deleted vertices and edges dynamically into the database until the graph size reached the size of the original dataset. Finally, render tests were performed on all eight datasets. The test results in Table 47 show that the constrained graph drawing framework rendered real-world large graph layouts in fewer than 20 seconds once the initial graph was stored in the database. The test results also demonstrated the constrained graph drawing framework can render multiple real-world large graph layouts simultaneously over the Internet, which is useful in a collaborative environment.

Table 47. Rendering results of eight real-world large datasets.

Graph name	Size	Size (including dummy vertices)	Constrained Render time (milliseconds)	Graphviz Render time (milliseconds)	DynaDAG Render time (milliseconds)
as-caida20040105	16,301	69,740	10,813	too long	too long
as-caida20040202	16,493	80,961	11,906	too long	too long
as-caida20040301	16,655	93,723	14,422	too long	too long
as-caida20040405	16,874	95,892	16,703	too long	too long
as-caida20040503	17,160	95,625	16,813	too long	too long
as-caida20040607	17,306	98,274	16,766	too long	too long
as-caida20040705	17,509	99,364	16,468	too long	too long
as-caida20040802	17,655	107,948	19,266	too long	too long

Similar to the static tests, all three datasets were utilized to conduct performance tests due to six dynamic operations. As dynamic operations require both retrieving and saving data from and to the database, three detailed performance tests were performed. The first performance test measured the I/O cost of data retrieval from the database to memory for each dynamic operation. The second performance test was run to measure the I/O cost of data insertion to the database, and the third performance test was run to measure the total run time of the constrained graph drawing framework for each dynamic operation. Three performance test results due to six dynamic operation are shown in Appendix C. The combined results of all six tests due to dynamic operations are shown in Figures 72, 73, and 74, respectively. Figure 72 shows the data retrieval run time of the constrained graph drawing framework, Figure 73 shows the time taken to save data into database, and Figure 74 shows the total run time of six dynamic operations, which combines the run times from Figures 72 and 73.

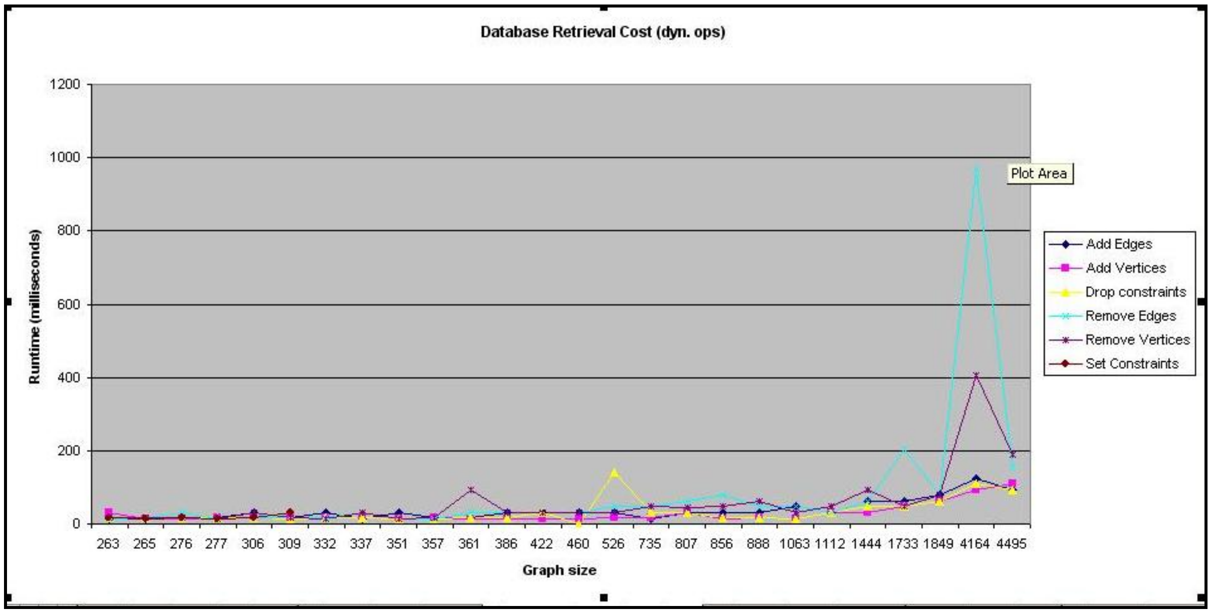


Figure 72. I/O cost due to data retrieval for six dynamic operations

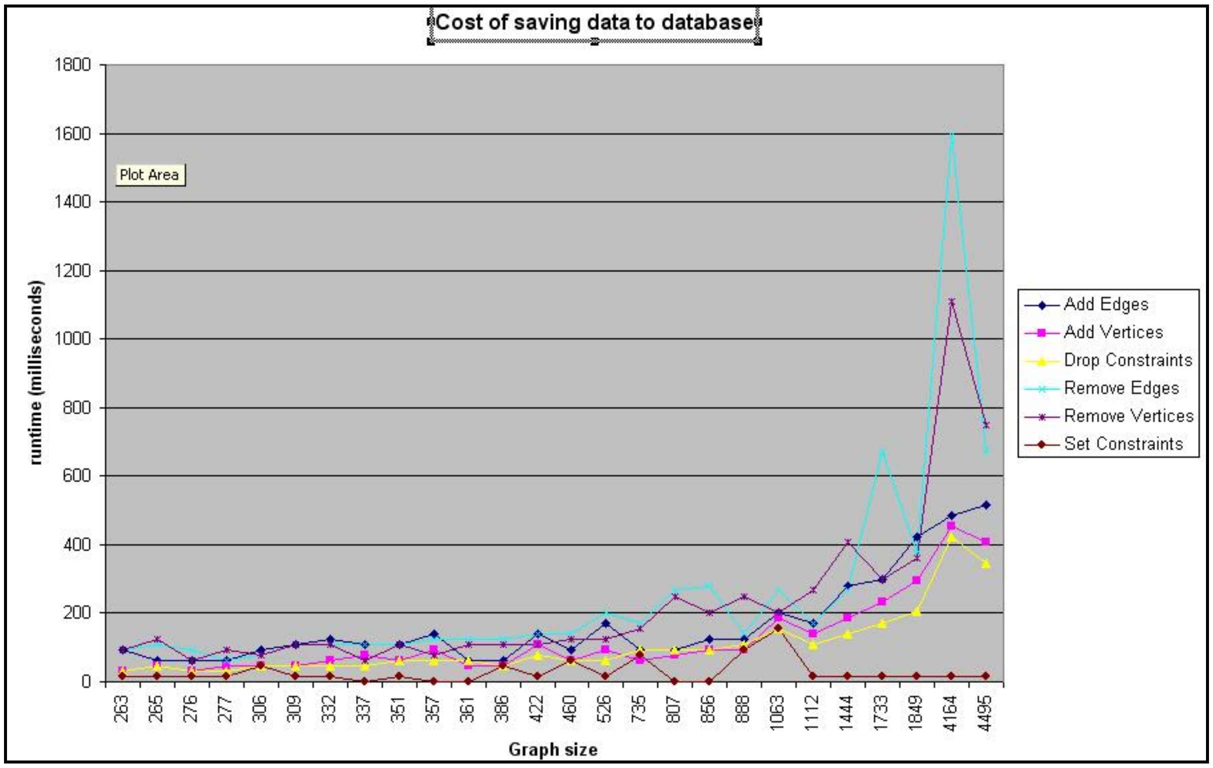


Figure 73. I/O cost due to database saving for six dynamic operations

The detailed run time analysis also shows that the Sugiyama heuristic employed in dynamic operations was fast, taking fewer than 0.5 seconds on average. Figure 73 shows that the bottleneck of dynamic operations was the step that saved data into the database. That the database saving operation was more expensive than data retrieval is due to the fact that the program copies the entire latest graph layout and creates a snapshot in the database once the graph model is updated. On the other hand, the data retrieval operation retrieves only data based on the impacted layers. This step could be improved by optimizing the relational data structure, which is discussed in Chapter 5.

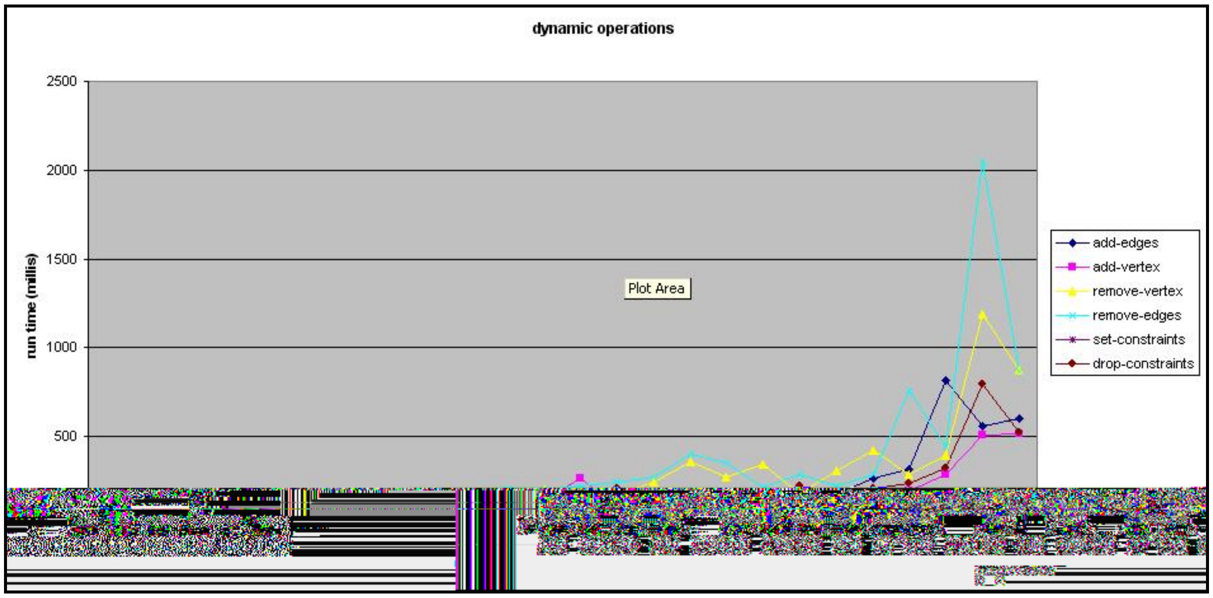


Figure 74. Performance of the constrained graph drawing framework dynamic operations

Because of the issue with the DynaDAG application, which did not return results when running against those datasets, the performance test results were compared against the performance of saving the entire graph into the database and the performance of Graphviz when rendering the same graph. The performance comparisons between the constrained graph drawing framework and Graphviz are shown in Figure 75. The chart shows that though the I/O cost of retrieval and insertion impact the overall performance of the constrained graph drawing framework dynamic operations, the size of the subgraph that was impacted by the operations helped to reduce their total run time. Thus, the run time of the constrained graph drawing framework was close to the run time of Graphviz and showed some performance improvement when graph sizes approached greater than 1,000 vertices. This result confirms that selecting only vertices and edges on impacted layers due to dynamic operations improves the performance of the constrained graph framework against the traditional static graph rendering frameworks.

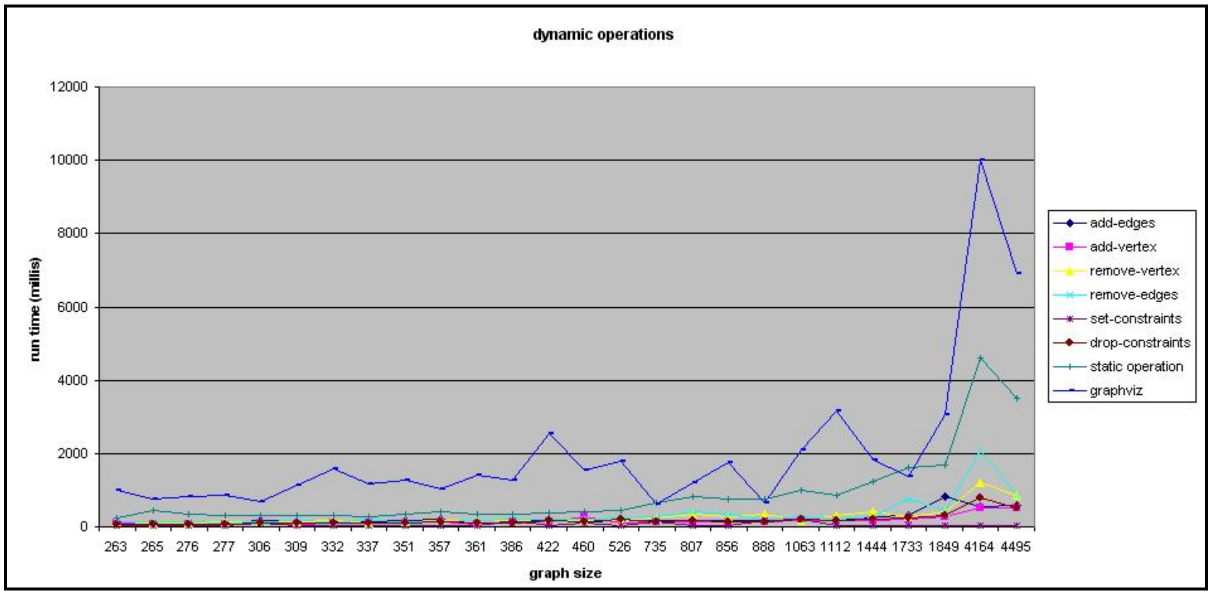


Figure 75. Performance comparisons of the constrained graph drawing framework dynamic operations vs. static graph saving operations and Graphviz

Asymptotic analysis for modified Sugiyama algorithms

The constrained graph drawing framework employed modified the Sugiyama algorithms.

Table 48 shows the asymptotic performance of algorithms that were utilized in the developed constrained graph drawing framework. Except for the modified barycenter algorithm, which had quadratic performance, the implementations of cycle removal and layer assignment algorithms had linear performance. The asymptotic analysis in Table 48 shows that the performance of the constrained graph drawing framework in the worst scenario was $\Theta(|C|^2)$, where C is the set of ordered constraints. However, in an average scenario the number of ordered constraints is much fewer than the number of vertices.

Table 48. Asymptotic analysis for modified Sugiyama algorithms.

Sugiyama step	Algorithm name	Asymptotic run time
Cycle removal	DFS	$O(\|V\|+\|E\|)$: V a set of vertices, E is a set of edges
Cycle removal	Greedy Cycle Removal	$O(\|V\|+\|E\|)$: V a set of vertices, E is a set of edges
Layer Assignment	Coffman-Graham	$O(\|V\|+\|E\|)$: V a set of vertices, E is a set of edges
Crossing reduction	Modified Barycenter	$O(\|C\|^2)$: C a set of constraints

Asymptotic analysis of I/O cost due to database operations

The constrained graph drawing framework employed a relational database for storing layout snapshots. Though the database enables the constrained graph drawing framework to decouple the graph visualization from the graph editor which improves the scalability by storing large datasets that cannot be stored in the computer main memory and helps to shift from desktop environment to distributed environment, the database does incur the overhead cost as the I/O cost is the dominant cost of a program run time. Two basic database operations that were utilized by the constrained graph drawing framework are insertion and retrieval. Insertion operation was executed while saving the layout snapshots from the computer memory to the database. The retrieval operations were called while retrieving impacted layers due to dynamic operations and retrieving graph layout snapshots for rendering. Because all six dynamic operations utilize the same function that retrieves vertices, edges, and constraints from the database based on the number of impacted layers, the database asymptotic analysis for retrieval operations can be applied to all six dynamic operations. The total I/O cost of insertion operation comprises eight insertion operations that are saving graph layout, graph snapshot, vertices, edges, vertex snapshots, edges snapshots, layers, and layer snapshots. Table 49 shows the I/O cost of each individual insertion operation, and Table 52 shows the total cost of all operations. The total I/O cost of retrieval due to dynamic operations comprises three individual transactions that are retrieving vertices that are assigned to the impacted layers, incident edges of those vertices, and constraints that are on the impacted layers. Let L′ be a set of $L_1, L_2, L_3, \ldots L_k$ layers where (1,2, .. k) are the impacted layers; Let V′ be a set of vertices where v $\in$ V′ are vertices that are assigned to impacted layers and E′ be a set of incident edges of v $\in$ V′. Let C′ =(u,v) be a set of ordered constraints where u, v are on the same layer and (u,v) $\in$ V′. Table 50 shows the I/O cost for each individual retrieval operation, and Table 52 shows the total cost of all operations due to a dynamic operation. The I/O cost of retrieval for graph layout rendering also comprises two transactions that are retrieving all vertex snapshots and edge snapshots as shown in Table 51.

Table 49. I/O cost of individual insertion operations.

Operation Type	Operation Name	Asymptotic run time
Insertion	Save graph information	O(1)
Insertion	Save snapshots	$O(\|V\|)$: V a set of vertices
Insertion	Save vertices	$O(\|V\|)$: V a set of vertices
Insertion	Save edges	$O(\|E\|)$: E a set of edges
Insertion	Save vertex snapshots	$O(\|V\|)$: V a set of vertices
Insertion	Save edge snapshots	$O(\|E\|)$: E a set of edges
Insertion	Save layers	$O(\|L\|)$: L a set of layers
Insertion	Save layer snapshots	$O(\|L\|)$: L a set of layers

Table 50. I/O cost of individual operations due to a dynamic operation.

Operation Type	Operation Name	Asymptotic run time
Retrieval	Get vertices by layer	$O(\lvert V' \rvert)$: V' a set of vertices on the impacted layers
Retrieval	Get edges by layers	$O(\lvert E' \rvert)$: E' a set of incident edges of V'
Retrieval	Get constraints by layers	$O(\lvert C' \rvert)$: C' a set of constraints

Table 51. I/O cost of retrieval of individual operations for rendering a graph layout.

Operation Type	Operation Name	Asymptotic run time
Retrieval	Get all vertices	$O(\lvert V \rvert)$: V a set of vertices
Retrieval	Get all edges	$O(\lvert E \rvert)$: E a set of edges

Table 52. Total I/O cost of database operations.

Step	Operation Type	Asymptotic run time
Save initial layout	Insertion	$O(\lvert V\rvert + \lvert E\rvert + \lvert L \rvert)$
Retrieve impacted layers due to dynamic operations	Retrieval	$O(\lvert V'\rvert + \lvert E'\rvert + \lvert C' \rvert)$
Retrieve snapshots for rendering a graph layout	Retrieval	$O(\lvert V\rvert + \lvert E\rvert)$

Chapter 5

Conclusions, Implications, Recommendations, and Summary

Conclusions

This research has shown that improved user experience for hierarchical graph drawing and viewing can be achieved by incrementally updating the graph layout and by storing graph layout snapshots into a database. This goal was achieved by incorporating two new features. The first feature improved layout stability by including ordered constraints of vertices on the same layer by extending the research of Forster (2004). The second feature improved the scalability of graph visualization by decoupling graph editing from the graph visualization components and storing graph layout snapshots into a relational database. Though using a relational database to store graph layouts incurred overhead due to the I/O cost of database operations, the following benefits flow from use of the database:

1. A graph drawing application could serve as "software as a service."
2. Enables graph drawing in collaborative environment.
3. Enables animation of the graph sequences.
4. Improves scalability.
5. Enables graph sequence analysis; for example, to view differences from one version to another.

A constrained graph drawing framework was developed based on the research of North and Woodhull (2001) and Forster (2004). The architecture of the constrained graph drawing framework was designed as client/server, which is similar to that of DynaDAG. In addition to supporting dynamic operations employed by DynaDAG, such as add vertex, add edge, remove vertex, and remove edge operations, the new framework implemented two new dynamic operations: (1) add ordered constraints and (2) remove ordered constraints. These two new dynamic operations were used to preserve layout stability. Unlike DynaDAG, the constrained graph drawing framework utilized a modified Sugiyama heuristic to generate graph layouts instead of using Network simplex. Another difference between DynaDAG and the new framework is that the constrained graph drawing framework was designed to separate graph editing from the visualization components and to store graph layouts and their snapshots into a database, which enhanced the scalability of the graph visualization and enabled the graph application to be deployed in a collaborative environment.

Three tests were performed. The first test was conducted to measure the effectiveness of the constrained graph drawing framework. The second test was run to measure its elegance, and the third test was conducted to measure the performance and scalability of the constrained graph drawing framework. The first test results showed that if the horizontal coordinate assignment factor is included from rendering, and if the overhead due to testing the Graphviz as a black box—which may have added some overhead to the overall run time—are excluded, the graph layouts generated by the constrained graph drawing framework are compatible with the layouts generated by Graphviz using the same dataset. The second test results showed that the constrained graph drawing framework preserved layer stability while updating graph layouts during dynamic operations. The third test was divided further into two tests, where the first test was run to measure the performance and scalability of the constrained graph drawing framework for generating static graph layouts and the second test was conducted to measure the performance and scalability of the constrained graph drawing framework during dynamic operations. The performance test results for generating static graph layouts showed that the constrained graph drawing framework performed better than Graphviz because the current implementation of the constrained graph drawing framework did not include a horizontal coordinate assignment algorithm in the Sugiyama heuristic and because the Graphviz test was run as a black box, which may have added some overhead to the calculated run time. Thus, the performances of the constrained graph drawing framework and Graphviz are within a few milliseconds of each other if the horizontal assignment is taken into account and if the runtime overhead due to the test mechanism is excluded. The scalability chart shows that as the graph size increased the constrained graph drawing framework performed reasonably well. The framework rendered a graph layout of about 10,000 vertices in a few seconds, which is acceptable in a collaborative environment.

Implications

By decoupling graph editing from the graph visualization component and utilizing a database to store the graph layout and its snapshots, this research presented many possibilities for graph viewing and graph analysis. One possibility is to move away from the traditional desktop environment to an online environment. HTML5 specifications are now mature, and newer versions of most browsers now support HTML5. Online interactive graph visualization such as graph animation is now possible if a sequence of layout snapshots is stored in a database. Another possibility for utilizing the storage of graph data in a database is to have graph versioning for enterprise process modeling. Furthermore, the problem of displaying very large graphs has been discussed in graph layout research. Searching and displaying subgraphs are now possible by retrieving subgraph snapshots from a database. Graph statistic functionality within the graph visualization has not been addressed widely, because most algorithms do not preserve the previous states of the graph. Storing graph data in a database makes it possible to have graph statistic features in graph visualization. The abstract model of aesthetic criteria can also be expanded to other types of graphs. For example, the orthogonal graph can also be implemented using this approach.

Recommendations

The constrained graph drawing framework developed in this research is still in an early state. The framework is still missing a number of important functions and features of a graph drawing system. Thus, many enhanced features should be developed for future versions of the constrained graph drawing framework before it could be used in real-world applications. An implementation of the horizontal coordinate assignment and support for Bezier curve and polyline drawing should improve the graph rendering engine and make it compatible with other real-world graph drawing and visualization applications. The run-time of the constrained graph drawing framework during dynamic operations can also be improved by optimizing the database saving function, which is the bottleneck of the constrained graph drawing framework performance. One possible enhancement to graph visualization is to include a search functionality to support the subgraph display functionality. Other improvements to the graph visualization component include a Google Maps-like cursor that allows the end user to move the graph to a location of interest by dragging with the mouse. This improvement should enhance user experience while viewing graph layouts on line. Several improvements in the graph editor component are possible. One improvement would be to have a graphical user interface that allows the user to modify the graph layout in a friendly graphical environment. Another improvement would be to allow the end user to add vertices without specifying a layer. The system should automatically assign a layer to the new vertex based on the given edge.

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 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.

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 goal of this dissertation was to develop a constrained graph drawing framework based on the work of North and Woodhull (2001) and to design a modified Sugiyama heuristic for solving the constrained one-sided crossing reduction problem based on the work by Forster (2004). The goal of the heuristic was to find a balance between the aesthetic criteria and minimizing the edge crossings.

This research first reviewed the aesthetic criteria for incremental graph layout. Then the research formulated an abstract model that represents those aesthetic criteria for incremental graph drawing and visualization. The objective of this abstract model was to define generic requirements for building an incremental graph drawing and visualization framework. The research then translated the abstract model into a concrete implementation of a modified Sugiyama heuristic, which was utilized by the developed constrained graph drawing framework. The enhancement of the modified Sugiyama was that it included ordered constraints of vertices on the layer, which preserves layout stability. A new constrained graph drawing framework was then developed. The architecture of the new framework is a client/server model in which clients communicate with a server through the HTTP and TCP protocols. Unlike DynaDAG, the newly developed framework completely decoupled the graph editor from the graph visualization process through the use of a relational database. Once receiving update operations from the graph editing component (client), the server first updates the layout utilizing the modified Sugiyama heuristic. Then it saves the updated layout and its snapshot into a relational database. The graph visualization component (client) retrieves the graph layout snapshot from the server and displays the graph layout on line. This process runs on separated threads asynchronously.

The research conducted three tests that measured the effectiveness, elegance, and efficiency (performance and scalability) of the developed constrained graph drawing framework. Due to the limitations of the graph visualization component, which does not implement the horizontal coordinate assignment and does not draw multilayer edges as polylines, the research defined three aesthetic criteria that were used to compare layouts generated using the same datasets. Using this scoring mechanism, the tests showed that layouts generated by the constrained graph drawing framework were compatible with layouts generated by Graphviz. Hence, the constrained graph drawing framework satisfied the effectiveness and elegance criteria.

To measure the efficiency of the constrained graph drawing framework in drawing static graphs, a performance test was conducted that compared the performance of the constrained graph drawing framework with that of Graphviz. As the current version of DynaDAG could not handle a large graph layout, to measure the efficiency of the constrained graph drawing framework in drawing graph layout during dynamic operations a simple performance was run and the result displayed to show the run time of the developed framework.

In summary, though many enhancements for the developed constrained graph drawing framework are possible, the research achieved the goals defined in Chapter 1. First, the research defined an abstract model representing aesthetic criteria for incremental graph layouts. Second, the research developed a constrained graph drawing framework based on the work of North and Woodhull (2001) that supports six dynamic operations. Third, the research implemented a modified Sugiyama heuristic in the constrained graph drawing framework by extending the work of Forster (2004). Fourth, the test results showed the developed constrained graph drawing framework satisfied all three aesthetic criteria for incremental graph drawing and visualization. In addition to the dissertation goals, the research demonstrated that although decoupling graph visualization from graph editing functionality—and utilizing a relational database to store graph layouts and their snapshots incurred additional overhead due to the I/O cost of database operations, the constrained graph drawing framework improves scalability and provides a foundation for graph editing applications in a collaborative environment. These features could be extended to support potential features such as graph animation, graph versioning, graph analysis, and drawing subgraphs for very large graphs in real time.

Appendix A

Modified Sugiyama Algorithms

TopologicalSortDFS source code

```
public class TopologicalSortDFS implements TopologicalSort {
 List<Vertex> list;
 List<Vertex> result;
 List<Vertex> sources;
 List<Vertex> sinks;

 public List<Vertex> sort(final List<Vertex> vertices) {
  list = new ArrayList<Vertex>();
  list.addAll(vertices);

  sources = new ArrayList<Vertex>();
  sinks = new ArrayList<Vertex>();
  result = new ArrayList<Vertex>();
  for (Iterator<Vertex> iter = list.iterator(); iter.hasNext();) {
   Vertex vertex = iter.next();
   // isolated vertices
   if (0 == vertex.getIndegree() && 0 == vertex.getOutdegree()) {
    sinks.add(vertex);
    iter.remove();
   }
   // sources
   else if (0 == vertex.getIndegree()) {
    sources.add(vertex);
    iter.remove();
   }
  }

  // result.addAll(sources);
  for (Vertex v : sources) {
   addChild(v);
  }

  if (!list.isEmpty()) {
   for (Vertex v : list) {
    addRemainder(v);
   }
  }
  result.addAll(sinks);
  return result;
 }

 private void addChild(Vertex v) {
  if (result.contains(v))
   return;
  result.add(v);
  int index = list.indexOf(v);
  if (index >= 0)
   list.remove(index);
  for (Vertex child : v.getChildren()) {
   addChild(child);
  }
 }

 private void addRemainder(Vertex v) {
  if (result.contains(v))
   return;
  result.add(v);
  for (Vertex child : v.getChildren()) {
   addRemainder(child);
  }
 }

}
```

TopologicalSortGreedy source code

```
public class TopologicalSortGreedy implements TopologicalSort {
 protected final Log logger = LogFactory.getLog(getClass());
 List<Vertex>        list;
 List<Vertex>        result;
 List<Vertex>        sources;
 List<Vertex>        sinks;
 List<Vertex>        vertices;

 public List<Vertex> sort(final List<Vertex> input) {
  list = new ArrayList<Vertex>();
  list.addAll(input);
  sinks = new ArrayList<Vertex>();
  sources = new ArrayList<Vertex>();
  // vertices = new ArrayList<Vertex>();

  Comparator<Vertex> reverse = new ReverseComparator();

  while (false == list.isEmpty()) {
   for (Iterator<Vertex> iter = list.iterator(); iter.hasNext();) {
    Vertex vertex = iter.next();
    // isolated vertices
    if (0 == (vertex.getOutdegree() - vertex.getCycleRemovalOutdegree())) {
     sinks.add(vertex);
     for (Vertex parent : vertex.getParents()) {
      parent.setCycleRemovalOutdegree(parent.getCycleRemovalOutdegree() + 1);
     }
     iter.remove();
    }
   }
   for (Iterator<Vertex> iter = list.iterator(); iter.hasNext();) {
    Vertex vertex = iter.next();
    // sources
    if (0 == (vertex.getIndegree() - vertex.getCycleRemovalIndegree())) {
     sources.add(vertex);
     for (Vertex child : vertex.getChildren()) {
      child.setCycleRemovalIndegree(child.getCycleRemovalIndegree() + 1);
     }
     iter.remove();
    }
   }
   if (false == list.isEmpty()) {
    List<Vertex> temp = new ArrayList<Vertex>();
    for (Vertex v : list) {
     temp.add(new Vertex(v.getId(), ((v.getOutdegree() - v
       .getCycleRemovalOutdegree()) - (v.getIndegree() - v
       .getCycleRemovalIndegree()))));
    }
    // sort vertices base on their delta in a reversed order

    Collections.sort(temp, reverse);
    Vertex vertex = temp.get(0);
    int index = list.indexOf(vertex);
    vertex = list.get(index);
    // logger.debug(vertex.getId());
    for (Vertex child : vertex.getChildren()) {
     child.setCycleRemovalIndegree(child.getCycleRemovalIndegree() + 1);
    }
    for (Vertex parent : vertex.getParents()) {
     parent.setCycleRemovalOutdegree(parent.getCycleRemovalOutdegree() + 1);
    }
    sources.add(vertex);
    list.remove(index);
   }
  }


  sources.addAll(sinks);
  return sources;
 }

 public static class ReverseComparator implements Comparator<Vertex> {
  public int compare(Vertex arg0, Vertex arg1) {
   if (arg1.value > arg0.value)
```

```
     return 1;
    else if (arg1.value == arg0.value)
     return 0;
    else
     return -1;
   }
  }

 }
```

LayerAssignmentTopDownImpl source code

```
public class LayerAssignmentTopDownImpl extends LayerAssignmentImpl {
 protected final Log    logger          = LogFactory.getLog(getClass());
 static LabelComparator labelComparator = new LabelComparator();
 List<Vertex>           result          = new ArrayList<Vertex>();
 /*
  * assign vertices into layers 1. sort vertices based on lexicographical order
  * starts from the sinks
  */
 public void assign(SimpleGraph graph, int max) throws Exception {
  Set<Vertex> assignedlist = new HashSet<Vertex>();
  List<Vertex> sortedlist = sort(graph);
  if (graph.getVertices().size() != sortedlist.size()) {
   throw new Exception("assigning layer: vertices do not match after sorting");
  }
  LayerDataStructure layers = graph.layers;
  layers.setMax(max);
  // add all source to the list
  for (Iterator<Vertex> iter = sortedlist.iterator(); iter.hasNext();) {
   Vertex v = iter.next();
   v.getParentLabels().clear();
   if (0 == v.getIndegree()) {
    layers.add(v);
    iter.remove();
    assignedlist.add(v);
    logger.debug("add " + v.getId() + " to layer: " + layers.getCurrentlayer());
   }
  }
  layers.nextLayer();
  // recursively assign remaining vertices into layers
  // stop when all vertices are assigned onto layers
  ReverseComparator comparator = new ReverseComparator();
  while (false == sortedlist.isEmpty()) {
   List<Vertex> locals = findUnassignedLayeredVertices(graph, assignedlist,
     sortedlist);
   if (false == locals.isEmpty()) {
    Collections.sort(locals, comparator);
    for (Vertex v : locals) {
     layers.add(v);
     assignedlist.add(v);
     sortedlist.remove(v);
    }
    locals.clear();
   }
   layers.nextLayer();
  }
  assignedlist.clear();
  addDummyVertices(graph);
  graph.getLayers().cleanupEmptyLayer();
 }

 List<Vertex> findUnassignedLayeredVertices(SimpleGraph graph,
   Set<Vertex> assignedlist, List<Vertex> sortedlist) throws Exception {
  List<Vertex> result = new ArrayList<Vertex>();
  LayerDataStructure layers = graph.layers;
  // find set R of vertices whose sinks have been assigned to layer
  for (Vertex v : sortedlist) {
   List parents = v.getParents();
   logger.debug(v.getId() + ": parents "
     + SimpleGraphUtil.printVertexSummary(parents));
   if (assignedlist.containsAll(parents)) {
    result.add(v);
    logger.debug("added vertex: " + v.getId() + " to assigned list");
   }
  }
  return result;
 }

}
```

CrossingReductionConstraintBaryCenter source code

```
public class CrossingReductionConstraintBaryCenter extends
  CrossingReductionBaryCenter {
 protected final Log logger = LogFactory.getLog(getClass());

 /**
  * reorder a layer(i) based on barycenter values
  */
 public void sortVerticesBasedonBarycenter(SimpleGraph graph, int layer,
   int nextlayer) throws Exception {
  BarycenterComparator comparator = new BarycenterComparator();

  List<Vertex> vertices = computeBarycenterValue(graph, layer, nextlayer);
  logger
    .debug("CrossingReductionConstraintBaryCenter before sorted" + vertices);
  Collections.sort(vertices, comparator);
  logger.debug("CrossingReductionConstraintBaryCenter after sorted" + vertices);

  Map<Vertex, Edge> violatedConstraints = findViolatedConstraints(graph,
    vertices, graph.getConstraints(), layer, nextlayer);
  if (!violatedConstraints.isEmpty()) {
   for (Vertex v : violatedConstraints.keySet()) {
    Edge e = violatedConstraints.get(v);
    vertices.remove(e.getSource());
    vertices.remove(e.getSink());
    vertices.add(v);
   }
   vertices = computeBarycenterValue(graph, layer, nextlayer);
   /**
    * sort the list based on bary center values
    */
   Collections.sort(vertices, comparator);
   for (Vertex v : violatedConstraints.keySet()) {
    Edge e = violatedConstraints.get(v);
    int index = vertices.indexOf(v);
    Vertex source = e.getSource();
    Vertex sink = e.getSink();
    source.setValue(v.getValue());
    sink.setValue(v.getValue() + 0.01f);
    source.setIndex(index);
    sink.setIndex(index + 1);
    logger.debug("add source and sink back " + source.getId() + ", "
      + sink.getId());
    vertices.remove(index);
    vertices.add(index, source);
    vertices.add(index + 1, sink);
   }
  }
  int pos = 0;
  for (Vertex v : vertices) {
   v.setIndex(pos++);
   graph.barycenter.add(v);
  }
 }
 Map<Vertex, Edge> findViolatedConstraints(SimpleGraph graph,
   List<Vertex> vertices, List<Edge> constraints, int layer, int nextLayer) {
  Map<Vertex, Edge> result = new HashMap<Vertex, Edge>();
  for (Edge e : constraints) {
   if (vertices.contains(e.getSource())) {
    Vertex source = vertices.get(vertices.indexOf(e.getSource()));
    Vertex sink = vertices.get(vertices.indexOf(e.getSink()));
    if (source.getValue() > sink.getValue()
      || source.getIndex() > sink.getIndex()) {
     Vertex v = createDummyVertex(graph, source, sink, layer, nextLayer);
     logger
       .debug("new dummy vertex created to compensate the violated constraints");
     Edge edge = new Edge(source, sink);
     result.put(v, edge);
    }
   }
  }
  return result;
 }
```

```
  private Vertex createDummyVertex(SimpleGraph graph, Vertex source,
    Vertex sink, int layer, int nextLayer) {
   int vertexNextValue = graph.getVertexMaxValue() + 1;
   Vertex v = new Vertex(vertexNextValue);
   int sourceDegree = (layer < nextLayer) ? source.getOutdegree() : source
     .getIndegree();
   int sinkDegree = (layer < nextLayer) ? sink.getOutdegree() : sink
     .getIndegree();
   v.setValue((source.getValue() * sourceDegree + sink.getValue() * sinkDegree)
     / (sourceDegree + sinkDegree));
   return v;
  }

}
```

GraphAction source code

```
public class GraphAction implements Action {
 protected final Log        logger = LogFactory.getLog(getClass());
 private ApplicationContext ctx;
 GraphDAO                   dao;
 GeneralJdbcDao             general;
 PojoDAOImpl                pojo;
 SimpleGraphService         service;
 CrossingReductionSweep     crossingReductionAlg;
 int                        iteration;

 public GraphAction(int maxIteration) {
  ctx = new FileSystemXmlApplicationContext("/WEB-INF/graph-servlet.xml");
  dao = (GraphDAO) ctx.getBean("graphDAO");
  general = (GeneralJdbcDao) ctx.getBean("generalDAO");
  service = (SimpleGraphService) ctx.getBean("simpleGraphPOJService");

  crossingReductionAlg = CrossingReductionSweepFactory.getAlg(
    CrossingReductionSweepFactory.DYNAMIC, maxIteration);
  iteration = maxIteration;
  pojo = new PojoDAOImpl();
 }

 public Object execute(String s) throws Exception {
  return execute(new StringReader(s));
 }

 public Object execute(Reader in) throws Exception {
  LexicalAnalyzer lexical = new DotLexicalAnalyzer(in);
  DotGraphParser parser = new DotGraphParser(lexical);
  Object result = null;
  AbstractExpression absyn = parser.parse();
  // logger.debug("executing command " + absyn.getExpressionType());
  if (AbstractExpression.GRAPH == absyn.getExpressionType()) {
   return executeGraph(absyn, parser);
  } else if (AbstractExpression.ADD_VERTEX == absyn.getExpressionType()) {
   result = executeAddVertexOperation(absyn, parser);
  } else if (AbstractExpression.REMOVE_VERTEX ==
absyn.getExpressionType()) {
   result = executeRemoveVertexOperation(absyn, parser);
  } else if (AbstractExpression.ADD_EDGE == absyn.getExpressionType()) {
   result = executeAddEdgesOperation(absyn, parser);
  } else if (AbstractExpression.REMOVE_EDGE == absyn.getExpressionType())
{
   result = executeRemoveEdgesOperation(absyn, parser);
  }

  else if (AbstractExpression.SET_MOVEMENT == absyn.getExpressionType()) {

  } else if (AbstractExpression.REMOVE_MOVEMENT ==
absyn.getExpressionType()) {

  } else if (AbstractExpression.SET_ORDER == absyn.getExpressionType()) {
    result = executeSetOrderConstraintsOperations(absyn, parser);
   } else if (AbstractExpression.REMOVE_ORDER == absyn.getExpressionType())
{
    result = executeRemoveOrderConstraintsOperations(absyn, parser);
   }
  return result;
 }

 SimpleGraph executeAddVertexOperation(AbstractExpression absyn,
    DotGraphParser parser) throws Exception {
  AddVerticesExp exp = (AddVerticesExp) absyn;
  List<Edge> edges = new ArrayList<Edge>();
  List<Vertex> vertices = new ArrayList<Vertex>();
  List<Edge> multi = new ArrayList<Edge>();

  String id = exp.getId();
  int newVertexId = new Integer(id).intValue();
  int lastImpactedLayer = exp.getLayer();
  int firstImpactedLayer = exp.getLayer();

  // posible next value for dummy vertices if any
  int vertexNextValue = general.getQueryForInt(
    "select max(vertex_id) from vertex where graph_name=?",
    new Object[] { exp.getGraphName() });

  vertexNextValue = (vertexNextValue > newVertexId) ? (vertexNextValue +
1)
    : (newVertexId + 1);
  logger.debug("next possible vertex id " + vertexNextValue);
  /**
   * simple test to check if the vertex is new or existing
   */
  VertexObject test = dao.getVertex(exp.getGraphName(), newVertexId);
  if (null != test) {
   throw new Exception("Vertex already exists..." + id);
  }

  if (exp.getEdges().isEmpty()) {
   throw new Exception("Missing required edges.. Please add edges..");
  }

  Vertex newVertex = new Vertex(newVertexId, 0);
  newVertex.setLayer(exp.getLayer());
  newVertex.setExisted(false);
  newVertex.setDummy(0);
  newVertex.setExisted(false);
  logger.debug("new vertex " + newVertex.getId() + " layer "
    + newVertex.getLayer());

  // add vertex into a list vertices
  vertices.add(newVertex);

  Vertex source = null;
  Vertex sink = null;
  for (EdgeAbstract edge : exp.getEdges()) {
   int sourceid = new Integer(edge.getSource()).intValue();
   int sinkid = new Integer(edge.getSink()).intValue();

   logger.debug("edge " + sourceid + ":" + sinkid);

   if (sourceid == sinkid) {
    throw new Exception("two cycle not supported " + sourceid + ": " +
sinkid);
   }

   if (sourceid == newVertexId) {
    source = newVertex;
    VertexObject vertex = dao.getVertex(exp.getGraphName(), sinkid);
    sink = new Vertex(vertex.getVertexId(), vertex.getBarycentric());
    sink.setIndex(vertex.getPosition());
    sink.setLayer(vertex.getLayer());
    sink.setDummy(vertex.getDummy());
    sink.setExisted(true);
   } else {
    sink = newVertex;
    VertexObject vertex = dao.getVertex(exp.getGraphName(), sourceid);
    source = new Vertex(vertex.getVertexId(), vertex.getBarycentric());
    source.setIndex(vertex.getPosition());
    source.setLayer(vertex.getLayer());
    source.setDummy(vertex.getDummy());
    source.setExisted(true);
   }
   // if layer(sink) < layer(source)
   // ==> likely create cycle
   // ==> needs to be reversed
   if (source.getLayer() > sink.getLayer()) {
    Vertex temp = source;
    source = sink;
    sink = temp;
   }

   lastImpactedLayer = Math.max(sink.getLayer(), lastImpactedLayer);
   firstImpactedLayer = Math.min(source.getLayer(), firstImpactedLayer);
   logger.debug("first layer " + firstImpactedLayer + " last layer "
     + lastImpactedLayer);

   addEdge(source, sink, vertices, edges, multi, vertexNextValue);
  }

  // get vertices and edges in these impacted layers
  long start2 = System.currentTimeMillis();
  SimpleGraph graph = getImpactedVerticesEdges(exp.getGraphName(),
    firstImpactedLayer, lastImpactedLayer, vertices, edges, multi);
  long end2 = System.currentTimeMillis();
  logger.info("get data from db takes: " + (end2 - start2));
  logger.debug("firstImpactedLayer " + firstImpactedLayer + ", max layer "
    + lastImpactedLayer + ", edges " + graph.getEdges().size() + ",
vertices "
    + graph.getVertices().size());

  // execute function here
  long start = System.currentTimeMillis();
  crossingReductionAlg.reduce(graph);
  long end1 = System.currentTimeMillis();
  System.out.println("constrained crossing reduction takes " + (end1 -
start));
  service.saveDynamicGraph(graph);
  long end = System.currentTimeMillis();
  System.out.println("save into db takes " + (end - end1));
  return graph;
 }

 SimpleGraph executeRemoveVertexOperation(AbstractExpression absyn,
   DotGraphParser parser) throws Exception {
  SimpleGraph graph = new SimpleGraph();
  AddVerticesExp exp = (AddVerticesExp) absyn;

  String id = exp.getId();
  int vertexId = new Integer(id).intValue();
  VertexObject vertexObject = dao.getVertex(exp.getGraphName(), vertexId);
  if (null == vertexObject) {
   logger.warn("vertex not found " + exp.getId());
   return graph;
  }

  int children = general.getQueryForInt(
    "select count(*) from edge where graph_name=? and head=? ", new
Object[] {
      exp.getGraphName(), new Integer(exp.getId()) });
  int parents = general.getQueryForInt(
    "select count(*) from edge where  graph_name=? and tail=?", new
Object[] {
      exp.getGraphName(), new Integer(exp.getId()) });

  int lastImpactedLayer = (children > 0) ? vertexObject.getLayer() + 1
    : vertexObject.getLayer();
  int firstImpactedLayer = (parents > 0) ? vertexObject.getLayer() - 1
    : vertexObject.getLayer();


  pojo.updateByQuery("delete from edge where graph_name='" +
exp.getGraphName()
    + "' and (head=" + id + " or tail=" + id + ")");

  pojo.updateByQuery("delete from vertex where graph_name='"
    + exp.getGraphName() + "' and vertex_id=" + id);

  pojo.updateByQuery("delete from order_constraint where graph_name='"
    + exp.getGraphName() + "' and (vertex1_id=" + id + " or vertex2_id=" +
id
    + ")");
  // get vertices and edges in these impacted layers
  graph = getImpactedVerticesEdges(exp.getGraphName(), firstImpactedLayer,
    lastImpactedLayer, new ArrayList<Vertex>(), new ArrayList<Edge>(),
    new ArrayList<Edge>());
  logger.debug("total vertices before dummy " +
graph.getVertices().size());

  crossingReductionAlg.reduce(graph);
  service.saveDynamicGraph(graph);
  return graph;
 }

 Object executeMovementConstraintsOperations(AbstractExpression absyn,
   DotGraphParser parser) {
  SimpleGraph graph = new SimpleGraph();
  SetConstraintExp exp = (SetConstraintExp) absyn;
  String graphName = exp.getGraphName();
  List<Integer> vertices = exp.getVertice();
  // execute function here
  if (AbstractExpression.REMOVE_MOVEMENT == absyn.getExpressionType()) {

  } else if (AbstractExpression.SET_MOVEMENT == absyn.getExpressionType())
{

  }
  return graph;
 }

 SimpleGraph executeAddEdgesOperation(AbstractExpression absyn,
   DotGraphParser parser) throws Exception {
  List<Vertex> vertices = new ArrayList<Vertex>();
  List<Edge> edges = new ArrayList<Edge>();
  List<Edge> multi = new ArrayList<Edge>();
  AddEdgesExp exp = (AddEdgesExp) absyn;

  int lastImpactedLayer = 0;
  int firstImpactedLayer = 0;

  Vertex source = null;
  Vertex sink = null;

  // posible next value for dummy vertices if any
  int vertexNextValue = general.getQueryForInt(
    "select max(vertex_id) from vertex where graph_name=?",
    new Object[] { exp.getGraphName() });
  vertexNextValue += 1;

  logger.debug("next possible vertex id " + vertexNextValue);

  for (EdgeAbstract edge : exp.getEdges()) {
   int sourceid = new Integer(edge.getSource()).intValue();
   int sinkid = new Integer(edge.getSink()).intValue();
```

```
    if (sourceid == sinkid) {
     throw new Exception("cycle not supported " + sourceid + ": " +
sinkid);
    }

    VertexObject temp1 = dao.getVertex(exp.getGraphName(), sourceid);
    VertexObject temp2 = dao.getVertex(exp.getGraphName(), sinkid);
    source = new Vertex(temp1);
    sink = new Vertex(temp2);

    // if layer(sink) < layer(source)
    // ==> likely create cycle
    // ==> needs to be reversed
    if (source.getLayer() > sink.getLayer()) {
     Vertex temp = source;
     source = sink;
     sink = temp;
    }

    lastImpactedLayer = Math.max(sink.getLayer(), lastImpactedLayer);
    firstImpactedLayer = Math.min(source.getLayer(), firstImpactedLayer);
    logger.info("first layer " + firstImpactedLayer + " last layer "
      + lastImpactedLayer);

    addEdge(source, sink, vertices, edges, multi, vertexNextValue++);
   }

   // get vertices and edges in these impacted layers
   SimpleGraph graph = getImpactedVerticesEdges(exp.getGraphName(),
     firstImpactedLayer, lastImpactedLayer, vertices, edges, multi);

   logger.debug("firstImpactedLayer " + firstImpactedLayer + ", max layer "
     + lastImpactedLayer + ", edges " + graph.getEdges().size() + ",
vertices "
     + graph.getVertices().size());
   // execute function here
   crossingReductionAlg.reduce(graph);
   service.saveDynamicGraph(graph);
   return graph;
  }

  SimpleGraph executeRemoveEdgesOperation(AbstractExpression absyn,
    DotGraphParser parser) throws Exception {
   AddEdgesExp exp = (AddEdgesExp) absyn;
   int lastImpactedLayer = 0;
   int firstImpactedLayer = 0;

   VertexObject source = null;
   VertexObject sink = null;
   SimpleGraph graph = null;

   for (EdgeAbstract edge : exp.getEdges()) {
    int sourceid = new Integer(edge.getSource()).intValue();
    int sinkid = new Integer(edge.getSink()).intValue();
    if (sourceid == sinkid) {
     throw new Exception("cycle not supported " + sourceid + ": " +
 sinkid);
    }

    source = dao.getVertex(exp.getGraphName(), sourceid);
    sink = dao.getVertex(exp.getGraphName(), sinkid);
    if (null != source && null != sink) {
     firstImpactedLayer = Math.min(source.getLayer(), firstImpactedLayer);
     lastImpactedLayer = Math.max(sink.getLayer(), lastImpactedLayer);
     EdgeObject e = dao.getEdge(exp.getGraphName(), sourceid, sinkid);
     if (1 == e.getMultilayer()) {
      dao.updateByQuery("delete from VertexObject where graphName='"
        + exp.getGraphName() + "' and dummy=1 and source=" + e.getSource()
        + " and sink=" + e.getSink());
      dao.updateByQuery("delete from EdgeObject where graphName='"
        + exp.getGraphName() + "' and dummy=1 and source=" + e.getSource()
        + " and sink=" + e.getSink());
     }

     dao.delete(e);

     // get vertices and edges in these impacted layers
     graph = getImpactedVerticesEdges(exp.getGraphName(),
 firstImpactedLayer,
       lastImpactedLayer, new ArrayList<Vertex>(), new ArrayList<Edge>(),
       new ArrayList<Edge>());

     logger.debug("firstImpactedLayer " + firstImpactedLayer + ", max layer
 "
       + lastImpactedLayer + ", edges " + graph.getEdges().size()
       + ", vertices " + graph.getVertices().size());

     crossingReductionAlg.reduce(graph);
     service.saveDynamicGraph(graph);
    }
   }
   return graph;
  }

  SimpleGraph executeSetOrderConstraintsOperations(AbstractExpression
 absyn,
    DotGraphParser parser) throws Exception {
   SimpleGraph graph = null;
   List constraintlist = new ArrayList();
   List snapshotlist = new ArrayList();
   SetOrderedConstraintExp exp = (SetOrderedConstraintExp) absyn;
   String graphName = exp.getGraphName();
   List<OrderedPairVertex> list = exp.getVertice();
   List<Edge> constraints = new ArrayList<Edge>();
   List<Edge> violated = new ArrayList<Edge>();
   int minLayer = 0;
   int maxLayer = 0;
```

```
   int currentversion = general.getLayoutSnapshotMaxVersion(graphName);

   int max = general.getQueryForInt(
     "select max(layer) from layer where graph_name=?",
     new Object[] { absyn.getGraphName() });

   for (OrderedPairVertex pair : list) {
    if (!pair.getVertex1().equals(pair.getVertex2())) {
     VertexObject v1 = dao.getVertex(graphName,
       new Integer(pair.getVertex1()).intValue());
     VertexObject v2 = dao.getVertex(graphName,
       new Integer(pair.getVertex2()).intValue());

     OrderConstraint constraint = new OrderConstraint();
     constraint.setGraphName(graphName);
     constraint.setLayer(v1.getLayer());
     constraint.setVertex1Id(v1.getVertexId());
     constraint.setVertex2Id(v2.getVertexId());
     constraintlist.add(constraint);

     OrderConstraintSnapshot snapshot = new OrderConstraintSnapshot();
     snapshot.setGraphName(graphName);
     snapshot.setGraphVersion(currentversion);
     snapshot.setLayer(v1.getLayer());
     snapshot.setVertex1Id(v1.getVertexId());
     snapshot.setVertex2Id(v2.getVertexId());
     snapshotlist.add(snapshot);

     minLayer = Math.min(minLayer, v1.getLayer());
     maxLayer = Math.max(maxLayer, v1.getLayer());

     Vertex vertex1 = new Vertex(v1);
     vertex1.setIndex(v1.getPosition());

     Vertex vertex2 = new Vertex(v2);
     vertex1.setIndex(v2.getPosition());

     Edge e = new Edge(vertex1, vertex2);
     e.setExisted(false);
     constraints.add(e);
     /**
      * constraint is violated
      */
     if (v1.getBarycentric() > v2.getBarycentric()
       || v1.getPosition() > v2.getPosition()) {
      violated.add(e);
     }
    } else {
     logger.warn("cycle found " + pair.getVertex1());
    }
   }
   if (!violated.isEmpty()) {
    if (maxLayer == minLayer) {
     if (0 == minLayer)
      maxLayer = minLayer + 1;
     else if (maxLayer == max) {
      minLayer = minLayer - 1;
     }
    }
    // get vertices and edges in these impacted layers
    graph = getImpactedVerticesEdges(exp.getGraphName(), minLayer,
 maxLayer,
      new ArrayList<Vertex>(), new ArrayList<Edge>(), new
 ArrayList<Edge>());

    graph.getConstraints().addAll(constraints);
    crossingReductionAlg.reduce(graph);
    service.saveDynamicGraph(graph);
   } else {
    dao.saveAll(constraintlist);
    dao.saveAll(snapshotlist);
   }
   return graph;
  }

  Object executeRemoveOrderConstraintsOperations(AbstractExpression absyn,
    DotGraphParser parser) throws Exception {
   List<OrderConstraint> temp = new ArrayList<OrderConstraint>();
   List<OrderConstraintSnapshot> temp1 = new
 ArrayList<OrderConstraintSnapshot>();
   SetOrderedConstraintExp exp = (SetOrderedConstraintExp) absyn;
   String graphName = exp.getGraphName();
   List<OrderedPairVertex> list = exp.getVertice();
   int minLayer = 0;
   int maxLayer = 0;
   int max = general.getQueryForInt(
     "select max(layer) from layer where graph_name=?",
     new Object[] { absyn.getGraphName() });

   int currentversion = general.getLayoutSnapshotMaxVersion(graphName);

   for (OrderedPairVertex pair : list) {
    VertexObject v1 = dao.getVertex(graphName,
      new Integer(pair.getVertex1()).intValue());
    VertexObject v2 = dao.getVertex(graphName,
      new Integer(pair.getVertex2()).intValue());

    OrderConstraint constraint = dao.getOrderConstraint(graphName, new
 Integer(
      pair.getVertex1()).intValue(), new
 Integer(pair.getVertex2()).intValue());
    minLayer = maxLayer = constraint.getLayer();
    dao.updateByQuery("delete from OrderConstraint where graphName='"
      + exp.getGraphName() + "'and  vertex1Id=" + pair.getVertex1()
      + " and vertex2Id=" + pair.getVertex2());
   }
   if (maxLayer == minLayer) {
    if (0 == minLayer)
```

```
    maxLayer = minLayer + 1;
   else if (maxLayer == max) {
    minLayer = minLayer - 1;
   }
  }

  // get vertices and edges in these impacted layers
  long start = System.currentTimeMillis();
  SimpleGraph graph = getImpactedVerticesEdges(exp.getGraphName(),
minLayer,
    maxLayer, new ArrayList<Vertex>(), new ArrayList<Edge>(),
    new ArrayList<Edge>());
  long end = System.currentTimeMillis();
  logger.debug("get data from db takes: " + (end - start));
  crossingReductionAlg.reduce(graph);
  service.saveDynamicGraph(graph);
  return graph;
 }

 /*
  * retrieve data from database and build the subgraph
  */
 public SimpleGraph getImpactedVerticesEdges(String graphname, int
minLayer,
   int maxLayer, List<Vertex> newvertices, List<Edge> newedges, List<Edge>
multi)
   throws Exception {
  SimpleGraph graph = new SimpleGraph();
  graph.setName(graphname);
  graph.setStartImpactedLayer(minLayer);
  LayerDataStructure layers = graph.getLayers();

  int maxlayer = Integer.MAX_VALUE;
  int lastLayer = general.getQueryForInt(
    "select max(layer) from layer where graph_name=?",
    new Object[] { graphname });
  layers.setMax(maxlayer);
  System.out.println("retrieve data start layer " + minLayer
    + " and end layer " + maxLayer);

  long start = System.currentTimeMillis();
  for (int i = minLayer; i <= maxLayer; i++) {
   /*
    * retrieve vertices from database
    */
   List<VertexObject> vertices = pojo.getVerticeByLayer(graphname, i);

   for (VertexObject v : vertices) {
    Vertex vertex = new Vertex(v);
    logger.debug("add vertex " + vertex + " to layer " + (i - minLayer));
    graph.addVertex(vertex);
    layers.add(vertex, (i - minLayer));
   }
  }
```

```
   long end = System.currentTimeMillis();
   // System.out.println("get vertices takes " + (end - start));

   String query = "select * from edge where graph_name='" + graphname
     + "' and start_layer>=" + minLayer + " and end_layer<=" + maxLayer;
   // System.out.println(query);
   start = System.currentTimeMillis();
   /*
    * retrieve edges from database
    */
   List<Object> list = pojo.getEdgesByQuery(query);
   end = System.currentTimeMillis();
   // System.out.println("get edges takes " + (end - start));

   logger.debug("# edges: " + list.size() + " graph name " + graphname);

   for (Object o1 : list) {
    EdgeObject edge = (EdgeObject) o1;
    if (false == graph.hasVertex(new Vertex(edge.getHead(), 0))) {
     throw new Exception("getImpactedVerticesEdges: vertex not found "
       + edge.getHead());
    }
    if (false == graph.hasVertex(new Vertex(edge.getTail(), 0))) {
     throw new Exception("getImpactedVerticesEdges: vertex not found "
       + edge.getTail());
    }
    Vertex head = graph.getVertices().get(
      graph.getVertices().indexOf(new Vertex(edge.getHead(), 0)));
    Vertex tail = graph.getVertices().get(
      graph.getVertices().indexOf(new Vertex(edge.getTail(), 0)));
    // logger.debug(edge);
    Edge e = new Edge(head, tail, edge.getDummy(), edge.getMultilayer());
    e.setTop(edge.getSource());
    e.setBottom(edge.getSink());
    e.setExisted(true);
    if (0 == edge.getMultilayer()) {
     graph.addEdge(e);
    } else {
     graph.getMulti().add(e);
    }
   }

   start = System.currentTimeMillis();
   /*
    * retrieve additional edges from database if applicable
    */
   if (minLayer > 0) {
    query = "select * from vertex where graph_name='" + graphname
      + "' and layer=" + (minLayer - 1) + " order by vertex_id";
    String getEdges = "select * from edge where graph_name='" + graphname
      + "' and start_layer=" + (minLayer - 1) + " and end_layer=" +
minLayer;
    List<VertexObject> parents = pojo.getVerticeByQuery(query);
    List<Object> tempedges = pojo.getEdgesByQuery(getEdges);
    for (Object o : tempedges) {
     EdgeObject e = (EdgeObject) o;
     int tailpos = graph.getVertices().indexOf(new Vertex(e.getTail()));
     if (tailpos < 0) {
      throw new Exception("getImpactedVerticesEdges: vertex not found "
        + e.getTail() + "[" + minLayer + "," + maxLayer + "]");
     }
     VertexObject temp = new VertexObject();
     temp.setGraphName(graphname);
     temp.setVertexId(e.getHead());
     int headpos = Collections.binarySearch(parents, temp);
     if (headpos < 0) {
      throw new Exception("getImpactedVerticesEdges: vertex not found "
        + e.getHead() + "[" + minLayer + "," + maxLayer + "]");
     }
     Vertex tempVertex = graph.getVertices().get(tailpos);
     tempVertex.getParents().add(new Vertex(temp));
    }
   }

   /*
    * retrieve additional vertices from database if applicable
    */
   if (maxLayer < lastLayer) {
    query = "select * from vertex where graph_name='" + graphname
      + "' and layer=" + (maxLayer + 1) + " order by vertex_id";
    String getEdges = "select * from edge where graph_name='" + graphname
      + "' and start_layer=" + (maxLayer) + " and end_layer=" + (maxLayer +
1);
    List<VertexObject> childrens = pojo.getVerticeByQuery(query);
    List<Object> tempedges = pojo.getEdgesByQuery(getEdges);
    for (Object o : tempedges) {
     EdgeObject e = (EdgeObject) o;
     int headpos = graph.getVertices().indexOf(new Vertex(e.getHead()));
     if (headpos < 0) {
      throw new Exception("getImpactedVerticesEdges: vertex not found "
        + e.getTail() + "[" + minLayer + "," + maxLayer + "]");
     }
     VertexObject temp = new VertexObject();
     temp.setGraphName(graphname);
     temp.setVertexId(e.getTail());
     int tailpos = Collections.binarySearch(childrens, temp);
     if (tailpos < 0) {
      throw new Exception("getImpactedVerticesEdges: vertex not found "
        + e.getTail() + "[" + minLayer + "," + maxLayer + "]");
     }
     Vertex tempVertex = new Vertex(childrens.get(tailpos));
     tempVertex.getChildren().add(new Vertex(temp));
    }
   }

   end = System.currentTimeMillis();
   System.out.println("get parents and children takes " + (end - start));
   /**
```

```
    * add constraints if any
    */

   query = "select * from order_constraint where graph_name='" + graphname
     + "' and layer <=" + maxLayer + " and layer>=" + minLayer;
   /*
    * retrieve constraints from database
    */
   List<Object> constraints = pojo.getConstraintsByQuery(query);

   for (Object o : constraints) {
    OrderConstraint c = (OrderConstraint) o;
    int sourceid = c.getVertex1Id();
    int sinkid = c.getVertex2Id();
    logger.debug("add ordered constraint: " + sourceid + ":" + sinkid);
    Vertex source = graph.getVertices().get(
      graph.getVertices().indexOf(new Vertex(sourceid, 0)));
    Vertex sink = graph.getVertices().get(
      graph.getVertices().indexOf(new Vertex(sinkid, 0)));
    Edge e = new Edge(source, sink);
    e.setExisted(true);
    graph.getConstraints().add(e);
   }

   // add vertices onto the graph
   for (Vertex v1 : newvertices) {
    graph.addVertex(v1);
    // add new vertex onto the layer
    graph.getLayers().add(v1, v1.getLayer() - minLayer);
   }

   // add all new edges onto the graph
   for (Edge e : newedges) {
    logger.debug("Add vertex operation add edge " + e.source + ", " +
 e.sink);
    graph.addEdge(e);
   }

   // add all multi edges onto the graph
   for (Edge e : multi) {
    logger.debug("Add vertex operation add multi edge " + e.source + ", "
      + e.sink);
    graph.getMulti().add(e);
   }

   return graph;
  }

  SimpleGraph executeGraph(AbstractExpression absyn, DotGraphParser parser)
    throws Exception {
   GraphExp exp = (GraphExp) absyn;
   SimpleGraph graph = new SimpleGraph();
   graph.name = exp.getGraphName();
   // logger.debug(graph.name + " node count " +
   // graph.getVertexCount());

   for (Statement stmt : exp.getStatements()) {
    if (stmt instanceof NodeStatement) {
     NodeStatement nodestatement = (NodeStatement) stmt;
     NodeId node = nodestatement.getNodeid();
     graph.addVertex(new Integer(node.getId()).intValue());
     logger.debug("add vertex " + node.getId());
    } else if (stmt instanceof EdgeStatement) {
     EdgeStatement edgestmt = (EdgeStatement) stmt;
     NodeId id = edgestmt.getNodeid();
     String source = id.getId();
     NodeId id2 = edgestmt.getEdgeRHS().getNodeId();
     String sink = id2.getId();
     if (!source.equals(sink)) {
      logger.debug("add edge " + source + ", " + sink);
      graph
        .addEdge(new Integer(source).intValue(), new
 Integer(sink).intValue());
     } else {
      logger.info("self loop found " + source);
     }
    }
   }
   logger.debug("total vertices before cycle removal "
     + graph.getVertices().size() + ", total edge before cycle removal "
     + graph.getEdges().size());

   GreedyCycleRemovalImpl alg = new GreedyCycleRemovalImpl();
   long start = System.currentTimeMillis();
   alg.reverseCycles(graph);
   long end = System.currentTimeMillis();
   logger.info("cycle removal step takes " + (double) (end - start) /
 1000f);

   logger.debug("total edges " + graph.getEdges().size());
   int max = calculateMaxVerticesPerLayer(graph.getVertices().size());
   logger.info("total vertices " + graph.getVertices().size() + ", layer
 max "
     + max);
   start = System.currentTimeMillis();
   LayerAssignment layerAssignmentAlg = LayerAssignmentFactory
     .getAlg(LayerAssignmentFactory.TOPDOWN);

   logger.info("start assign vertices into layer...");
   layerAssignmentAlg.assign(graph, max);
   end = System.currentTimeMillis();

   logger.info("assign step takes " + (double) (end - start) / 1000f);
   logger.info("start reduce edge crossings ...");
   start = System.currentTimeMillis();

   CrossingReductionSweep sweep = CrossingReductionSweepFactory.getAlg(
     CrossingReductionSweepFactory.STD, 5);
   sweep.reduce(graph);
   end = System.currentTimeMillis();
   logger
     .info("crossing reduction step takes " + (double) (end - start) /
 1000f);

   logger.info("start save initial graph into databse....");
   start = System.currentTimeMillis();
   service.saveInitialGraph(graph);
   end = System.currentTimeMillis();
   logger.info("save data into relational database step takes "
     + (double) (end - start) / 1000f);

   return graph;
  }

  private void addEdge(Vertex source, Vertex sink, List<Vertex> vertices,
    List<Edge> edges, List<Edge> multi, int vertexNextValue) {
   // find and remove edges that cross more than one layer
   if ((sink.getLayer() - source.getLayer()) > 1) {
    logger.debug("not propered layers " + source + ", " + sink);

    Vertex newsource = source;
    Vertex newsink = sink;

    while (sink.getLayer() > newsource.getLayer() + 1) {
     newsink = new Vertex(vertexNextValue++, 0);
     newsink.setLayer(newsource.getLayer() + 1);
     newsink.setDummy(1);
     newsink.setSource(source.getId());
     newsink.setSink(sink.getId());
     logger.debug("add new dummy vertex " + newsink.getId());
     // add dummy vertex into the list of new vertices
     if (false == vertices.contains(newsink)) {
      vertices.add(newsink);
     }

     Edge newedge = new Edge(newsource, newsink, Edge.DUMMY);
     newedge.setTop(source.getId());
     newedge.setBottom(sink.getId());
     newedge.setExisted(false);
     edges.add(newedge);
     newsource = newsink;
    }
    // add last edge
    newsink = sink;
    Edge newedge = new Edge(newsource, newsink, Edge.DUMMY);
    newedge.setTop(source.getId());
    newedge.setBottom(sink.getId());
    newedge.setExisted(false);
    edges.add(newedge);
    multi.add(new Edge(source, sink, 0, 1));// add multi edges
   } else {
    edges.add(new Edge(source, sink));
   }
  }

  public static int calculateMaxVerticesPerLayer(int total) {
    return Integer.MAX_VALUE;
  }

}
```

Appendix B

Test Files in DOT Format

Third.dot file dataset that was used in the first visualization test

```
digraph "/home/mvinni/o/jspin411/tmp_t/thirdabbrev" {
0;
1;
2;
3;
4;
5;
6;
7;
0 -> 1;
1 -> 2;
2 -> 0;
2 -> 3;
3 -> 4;
4 -> 5;
5 -> 5;
5 -> 5;
4 -> 6;
6 -> 7;
7 -> 1;
6 -> 0;
1 -> 5;
0 -> 4;
}
```

Second.dot file dataset that was used in the first visualization test

```
digraph "/home/mvinni/o/jspin411/tmp t/secondabbrev" {
0;
1;
2;
3;
4;
5;
6;
7;
8;
0 -> 1;
1 -> 2;
2 -> 0;
1 -> 3;
3 -> 4;
4 -> 5;
5 -> 3;
5 -> 6;
6 -> 0;
4 -> 7;
7 -> 6;
7 -> 2;
3 -> 8;
8 -> 7;
8 -> 1;
0 -> 5;
}
```

org-parent-child-conversion-11.dot file dataset that was used in the first visualization test

```
digraph "org-parent-child-conversion-11" {
6669;
6670;
6672;
6673;
6674;
6676;
6677;
6678;
6679;
6680;
6671;
6675;
6669 -> 6670;
6669 -> 6672;
6669 -> 6673;
6669 -> 6674;
6669 -> 6676;
6669 -> 6677;
6669 -> 6678;
6669 -> 6679;
6669 -> 6680;
6670 -> 6671;
6674 -> 6675;
}
```

process-parent-child-conversion-16 file dataset that was used in the first visualization test

```
digraph "process-parent-child-conversion-16_0" {
1564;
1565;
1865;
2765;
507;
1866;
3199;
508;
1118;
1234;
1236;
1240;
1241;
1243;
1245;
1286;
1734;
1564 -> 1565;
1564 -> 1865;
1564 -> 2765;
1564 -> 507;
1865 -> 1866;
1865 -> 3199;
507 -> 508;
507 -> 1118;
507 -> 1234;
507 -> 1236;
507 -> 1240;
507 -> 1241;
507 -> 1243;
507 -> 1245;
507 -> 1286;
507 -> 1734;
}
```

org-parent-child-conversion-23_0.dot file dataset that was used in the first visualization test

```
digraph "org-parent-child-conversion-23_0" {
264;
265;
266;
267;
840;
403;
841;
842;
3311;
405;
406;
407;
408;
409;
667;
10079;
10080;
404;
464;
465;
466;
467;
468;
469;
264 -> 265;
264 -> 266;
264 -> 267;
264 -> 840;
264 -> 403;
264 -> 841;
264 -> 842;
264 -> 3311;
265 -> 405;
265 -> 406;
265 -> 407;
265 -> 408;
265 -> 409;
265 -> 667;
409 -> 10079;
409 -> 10080;
403 -> 404;
403 -> 464;
403 -> 465;
403 -> 466;
403 -> 467;
403 -> 468;
403 -> 469;
}
```

org-parent-child-conversion-12_33.dot file that was used in the second visualization test

```
digraph "org-parent-child-conversion-12_33" {
8478;
8479;
8482;
8486;
8490;
8480;
8481;
8483;
8484;
8485;
8487;
8488;
8489;
8478 -> 8479;
8478 -> 8482;
8478 -> 8486;
8478 -> 8490;
8479 -> 8480;
8479 -> 8481;
8482 -> 8483;
8482 -> 8484;
8482 -> 8485;
8486 -> 8487;
8486 -> 8488;
8486 -> 8489;
}
```

org-parent-child-conversion-263-dynamic.txt test file for dynamic operation tests

```
add vertices "org-parent-child-conversion-263" 2014 1 { 777->2014 }
add edges "org-parent-child-conversion-263" { 1214->2014 }
remove vertices "org-parent-child-conversion-263" 1948
remove edges "org-parent-child-conversion-263" { 1109 -> 1119  }
set order "org-parent-child-conversion-263" { 1918 < 1321  }
drop order "org-parent-child-conversion-263" { 1918 < 1321  }
```

org-parent-child-conversion-265-dynamic.txt test file for dynamic tests

```
add vertices "org-parent-child-conversion-265" 9706 1 { 31->9706 }
add edges "org-parent-child-conversion-265" { 8283->9706 }
remove vertices "org-parent-child-conversion-265" 225
remove edges "org-parent-child-conversion-265" { 8267 -> 8269  }
set order "org-parent-child-conversion-265" { 227 < 226  }
drop order "org-parent-child-conversion-265" { 227 < 226  }
```

org-parent-child-conversion-276-dynamic.txt test file for dynamic tests

```
add vertices "org-parent-child-conversion-276" 3953 1 { 70->3953 }
add edges "org-parent-child-conversion-276" { 476->3953 }
remove vertices "org-parent-child-conversion-276" 470
remove edges "org-parent-child-conversion-276" { 731 -> 3457  }
set order "org-parent-child-conversion-276" { 558 < 374  }
drop order "org-parent-child-conversion-276" { 558 < 374  }
```

org-parent-child-conversion-277-dynamic.txt test file for dynamic tests

```
add vertices "org-parent-child-conversion-277" 10104 1 { 5853->10104 }
add edges "org-parent-child-conversion-277" { 6353->10104 }
remove vertices "org-parent-child-conversion-277" 6846
remove edges "org-parent-child-conversion-277" { 6013 -> 6123  }
set order "org-parent-child-conversion-277" { 6349 < 5517  }
drop order "org-parent-child-conversion-277" { 6349 < 5517  }
```

org-parent-child-conversion-306-dynamic.txt test file for dynamic tests

```
add vertices "org-parent-child-conversion-306" 10340 1 { 644->10340 }
add edges "org-parent-child-conversion-306" { 2345->10340 }
remove vertices "org-parent-child-conversion-306" 2033
remove edges "org-parent-child-conversion-306" { 2453 -> 2460  }
set order "org-parent-child-conversion-306" { 2032 < 2020  }
drop order "org-parent-child-conversion-306" { 2032 < 2020  }
```

org-parent-child-conversion-309-dynamic.txt test file for dynamic tests

```
add vertices "org-parent-child-conversion-309" 2599 1 { 342->2599 }
add edges "org-parent-child-conversion-309" { 2081->2599 }
remove vertices "org-parent-child-conversion-309" 1736
remove edges "org-parent-child-conversion-309" { 1262 -> 1267  }
set order "org-parent-child-conversion-309" { 2085 < 2084  }
drop order "org-parent-child-conversion-309" { 2085 < 2084  }
```

org-parent-child-conversion-351-dynamic.txt test file for dynamic tests

```
add vertices "org-parent-child-conversion-351" 5165 1 { 49->5165 }
add edges "org-parent-child-conversion-351" { 4825->5165 }
remove vertices "org-parent-child-conversion-351" 4884
remove edges "org-parent-child-conversion-351" { 57 -> 4831  }
set order "org-parent-child-conversion-351" { 58 < 57  }
drop order "org-parent-child-conversion-351" { 58 < 57  }
```

org-parent-child-conversion-361-dynamic.txt test file for dynamic tests

```
add vertices "org-parent-child-conversion-361" 2014 1 { 243->2014 }
add edges "org-parent-child-conversion-361" { 1089->2014 }
remove vertices "org-parent-child-conversion-361" 1523
remove edges "org-parent-child-conversion-361" { 1934 -> 1935  }
set order "org-parent-child-conversion-361" { 777 < 1497  }
drop order "org-parent-child-conversion-361" { 777 < 1497  }
```

org-parent-child-conversion-386-dynamic.txt test file for dynamic tests

```
add vertices "org-parent-child-conversion-386" 10339 1 { 633->10339 }
add edges "org-parent-child-conversion-386" { 8674->10339 }
remove vertices "org-parent-child-conversion-386" 8561
remove edges "org-parent-child-conversion-386" { 8100 -> 8104  }
set order "org-parent-child-conversion-386" { 8693 < 8591  }
drop order "org-parent-child-conversion-386" { 8693 < 8591  }
```

org-parent-child-conversion-460-dynamic.txt test file for dynamic tests

```
add vertices "org-parent-child-conversion-460" 10366 1 { 360->10366 }
add edges "org-parent-child-conversion-460" { 10347->10366 }
remove vertices "org-parent-child-conversion-460" 9244
remove edges "org-parent-child-conversion-460" { 9784 -> 9786  }
set order "org-parent-child-conversion-460" { 9430 < 9426  }
drop order "org-parent-child-conversion-460" { 9430 < 9426  }
```

process-parent-child-conversion-332-dynamic.txt test file for dynamic tests

```
add vertices "process-parent-child-conversion-332" 3228 1 { 101->3228 }
add edges "process-parent-child-conversion-332" { 55->3228 }
remove vertices "process-parent-child-conversion-332" 889
remove edges "process-parent-child-conversion-332" { 149 -> 578  }
set order "process-parent-child-conversion-332" { 332 < 200  }
drop order "process-parent-child-conversion-332" { 332 < 200  }
```

process-parent-child-conversion-337-dynamic.txt test file for dynamic tests

```
add vertices "process-parent-child-conversion-337" 3221 1 { 48->3221 }
add edges "process-parent-child-conversion-337" { 2253->3221 }
remove vertices "process-parent-child-conversion-337" 1176
remove edges "process-parent-child-conversion-337" { 349 -> 2495  }
set order "process-parent-child-conversion-337" { 3200 < 3204  }
drop order "process-parent-child-conversion-337" { 3200 < 3204  }
```

process-parent-child-conversion-357-dynamic.txt test file for dynamic tests

```
add vertices "process-parent-child-conversion-357" 3231 1 { 19->3231 }
add edges "process-parent-child-conversion-357" { 1812->3231 }
remove vertices "process-parent-child-conversion-357" 862
remove edges "process-parent-child-conversion-357" { 203 -> 1716  }
set order "process-parent-child-conversion-357" { 1248 < 1149  }
drop order "process-parent-child-conversion-357" { 1248 < 1149  }
```

process-parent-child-conversion-422-dynamic.txt test file for dynamic tests

```
add vertices "process-parent-child-conversion-422" 3229 1 { 1->3229 }
add edges "process-parent-child-conversion-422" { 2650->3229 }
remove vertices "process-parent-child-conversion-422" 3217
remove edges "process-parent-child-conversion-422" { 23 -> 628  }
set order "process-parent-child-conversion-422" { 2637 < 2631  }
drop order "process-parent-child-conversion-422" { 2637 < 2631  }
```

org-parent-child-conversion-735-dynamic.txt test file for dynamic tests

```
add vertices "org-parent-child-conversion-735" 10294 1 { 658->10294 }
add edges "org-parent-child-conversion-735" { 5131->10294 }
remove vertices "org-parent-child-conversion-735" 5910
remove edges "org-parent-child-conversion-735" { 6657 -> 6659  }
set order "org-parent-child-conversion-735" { 6385 < 6383  }
drop order "org-parent-child-conversion-735" { 6385 < 6383  }
```

org-parent-child-conversion-807-dynamic.txt test file for dynamic tests

```
add vertices "org-parent-child-conversion-807" 4976 1 { 2919->4976 }
add edges "org-parent-child-conversion-807" { 3434->4976 }
remove vertices "org-parent-child-conversion-807" 3098
remove edges "org-parent-child-conversion-807" { 3974 -> 3977  }
set order "org-parent-child-conversion-807" { 3506 < 3495  }
drop order "org-parent-child-conversion-807" { 3506 < 3495  }
```

org-parent-child-conversion-856-dynamic.txt test file for dynamic tests

```
add vertices "org-parent-child-conversion-856" 10350 1 { 656->10350 }
add edges "org-parent-child-conversion-856" { 6153->10350 }
remove vertices "org-parent-child-conversion-856" 5196
remove edges "org-parent-child-conversion-856" { 5712 -> 5716  }
set order "org-parent-child-conversion-856" { 5901 < 5898  }
drop order "org-parent-child-conversion-856" { 5901 < 5898  }
```

org-parent-child-conversion-888-dynamic.txt test file for dynamic tests

```
add vertices "org-parent-child-conversion-888" 10296 1 { 664->10296 }
add edges "org-parent-child-conversion-888" { 7771->10296 }
remove vertices "org-parent-child-conversion-888" 6303
remove edges "org-parent-child-conversion-888" { 7765 -> 7781  }
set order "org-parent-child-conversion-888" { 6568 < 6562  }
drop order "org-parent-child-conversion-888" { 6568 < 6562  }
```

process-parent-child-conversion-526-dynamic.txt test file for dynamic tests

```
add vertices "process-parent-child-conversion-526" 3237 1 { 76->3237 }
add edges "process-parent-child-conversion-526" { 661->3237 }
remove vertices "process-parent-child-conversion-526" 2118
remove edges "process-parent-child-conversion-526" { 288 -> 929  }
set order "process-parent-child-conversion-526" { 873 < 629  }
drop order "process-parent-child-conversion-526" { 873 < 629  }
```

org-parent-child-conversion-1063-dynamic.txt test file for dynamic tests

```
add vertices "org-parent-child-conversion-1063" 10380 1 { 73->10380 }
add edges "org-parent-child-conversion-1063" { 8720->10380 }
remove vertices "org-parent-child-conversion-1063" 8721
remove edges "org-parent-child-conversion-1063" { 8965 -> 8969  }
set order "org-parent-child-conversion-1063" { 161 < 75  }
drop order "org-parent-child-conversion-1063" { 161 < 75  }
```

org-parent-child-conversion-1444-dynamic.txt test file for dynamic tests

```
add vertices "org-parent-child-conversion-1444" 10345 1 { 76->10345 }
add edges "org-parent-child-conversion-1444" { 169->10345 }
remove vertices "org-parent-child-conversion-1444" 2545
remove edges "org-parent-child-conversion-1444" { 159 -> 438  }
set order "org-parent-child-conversion-1444" { 2980 < 2918  }
drop order "org-parent-child-conversion-1444" { 2980 < 2918  }
```

org-parent-child-conversion-1733-dynamic.txt test file for dynamic tests

```
add vertices "org-parent-child-conversion-1733" 10342 1 { 35->10342 }
add edges "org-parent-child-conversion-1733" { 2822->10342 }
remove vertices "org-parent-child-conversion-1733" 1451
remove edges "org-parent-child-conversion-1733" { 1209 -> 1211  }
set order "org-parent-child-conversion-1733" { 651 < 649  }
drop order "org-parent-child-conversion-1733" { 651 < 649  }
```

org-parent-child-conversion-4164-dynamic.txt test file for dynamic tests

```
add vertices "org-parent-child-conversion-4164" 10354 1 { 33->10354 }
add edges "org-parent-child-conversion-4164" { 9157->10354 }
remove vertices "org-parent-child-conversion-4164" 8124
remove edges "org-parent-child-conversion-4164" { 5606 -> 5613  }
set order "org-parent-child-conversion-4164" { 661 < 660  }
drop order "org-parent-child-conversion-4164" { 661 < 660  }
```

process-parent-child-conversion-1112-dynamic.txt test file for dynamic tests

```
add vertices "process-parent-child-conversion-1112" 3259 1 { 11->3259 }
add edges "process-parent-child-conversion-1112" { 2938->3259 }
remove vertices "process-parent-child-conversion-1112" 2890
remove edges "process-parent-child-conversion-1112" { 101 -> 1108  }
set order "process-parent-child-conversion-1112" { 1615 < 1468  }
drop order "process-parent-child-conversion-1112" { 1615 < 1468  }
```

process-parent-child-conversion-1849-dynamic.txt test file for dynamic tests

```
add vertices "process-parent-child-conversion-1849" 3276 1 { 118->3276 }
add edges "process-parent-child-conversion-1849" { 2958->3276 }
remove vertices "process-parent-child-conversion-1849" 311
remove edges "process-parent-child-conversion-1849" { 93 -> 407  }
set order "process-parent-child-conversion-1849" { 760 < 722  }
drop order "process-parent-child-conversion-1849" { 760 < 722  }
```

process-parent-child-conversion-4495-dynamic.txt test file for dynamic tests

```
add vertices "process-parent-child-conversion-4495" 3564 1 { 8->3564 }
add edges "process-parent-child-conversion-4495" { 658->3564 }
remove vertices "process-parent-child-conversion-4495" 1051
remove edges "process-parent-child-conversion-4495" { 48 -> 1222  }
set order "process-parent-child-conversion-4495" { 1602 < 3539  }
drop order "process-parent-child-conversion-4495" { 1602 < 3539  }
```

Main function of the first performance test program

```
public void testAll() throws Exception {
  action = new MockGraphAction(1);;
  testMany(testfolder + "small-data-set", resultfolder, "-result");
  testMany(testfolder + "medium-data-set", resultfolder, "-result");
  testMany(testfolder + "large-data-set", resultfolder, "-result");
 }

 public void testMany(String dir, String diroutput, String suffix)
   throws Exception {
  File file = new File(dir);
  File[] files = file.listFiles(new FileFilter() {
   public boolean accept(File f) {
    return f.isFile() && f.getName().endsWith(".dot");
   }
  });
  File outputfile = new File(diroutput + file.getName() + suffix + ".txt");
  BufferedWriter writer = new BufferedWriter(new FileWriter(outputfile));
  writer.write("Graph name \t size \t dynamic \t graphviz\n");
  for (int i = 0; i < files.length; i++) {
   String name = ApplicationUtil.getNameWithoutExtension(files[i].getName());
   String size = name.substring(name.lastIndexOf("-") + 1);
   delete(name);
   long taken = testGraphVizPerformance(files[i]);
   long taken1 = testConstrainedFrameworkPerformance(files[i]);
   writer.write(name + "\t" + size + "\t" + taken1 + "\t" + taken + "\n");
   System.out.println(name + "\t" + size + "\t" + taken1 + "\t" + taken + "\n");
  }
  writer.flush();
  writer.close();

 }
```

Main function of the second performance test program

```
public void testAll() throws Exception {
  action = new MockGraphAction2(1);
  testMany(testfolder + "small-data-set", resultfolder, "-result2");
  testMany(testfolder + "medium-data-set", resultfolder, "-result2");
  testMany(testfolder + "large-data-set", resultfolder, "-result2");

 }
```

Main function of the test program that measures the I/O cost of data retrieval for rendering

```
public static void testAll(String dir, String diroutput, String suffix)
   throws Exception {
  ApplicationContext ctx = new FileSystemXmlApplicationContext(
    "/WEB-INF/graph-servlet.xml");
  SimpleGraphService service = (SimpleGraphService) ctx
    .getBean("simpleGraphService");
  GeneralJdbcDao general = (GeneralJdbcDao) ctx.getBean("generalDAO");
  File file = new File(dir);
  File[] files = file.listFiles(new FileFilter() {
   public boolean accept(File f) {
    return f.isFile() && f.getName().endsWith(".dot")
      && (f.getName().indexOf("graphviz-result") == -1);
   }
  });
  File outputfile = new File(diroutput + file.getName() + suffix + ".txt");
  BufferedWriter writer = new BufferedWriter(new FileWriter(outputfile));
  writer.write("Graph name \t size \t dynamic \t graphviz\n");
  for (int i = 0; i < files.length; i++) {
   String name = ApplicationUtil.getNameWithoutExtension(files[i].getName());
   String size = name.substring(name.lastIndexOf("-") + 1);
   // System.out.println("start render graph [" + name + "]");
   String[] localgraphs = service.getGraphsnapshots(name);
   String[] tokens = localgraphs[0].split(":");
   String graphname = tokens[0];
   String version = tokens[1];
   long start = System.currentTimeMillis();
   service.generateLayoutShapes(graphname, new Integer(version).intValue());
   long end = System.currentTimeMillis();
   long taken = (end - start);
   writer.write(name + "\t" + size + "\t" + taken + "\t" + "\n");
   System.out.println(name + "\t" + size + "\t" + taken + "\t" + "\n");
  }
  writer.flush();
  writer.close();

 }
```

Main function of the rendering performance test program

```
public static void testAll(String dir, String diroutput, String suffix)
   throws Exception {
  ApplicationContext ctx = new FileSystemXmlApplicationContext(
    "/WEB-INF/graph-servlet.xml");
  SimpleGraphService service = (SimpleGraphService) ctx
    .getBean("simpleGraphService");
  GeneralJdbcDao general = (GeneralJdbcDao) ctx.getBean("generalDAO");
  String[] graphs = service.getAllGraphVersions();
  StringBuffer buffer = new StringBuffer();
  String first = graphs[0];
  for (int i = 0; i < graphs.length; i++) {
   buffer.append(graphs[i] + ",");
  }
  String[] tokens = first.split(":");
  String graphname = tokens[0];
  String version = tokens[1];

  JFrame frame = new JFrame("graph viewer");
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  String layout = buffer.toString();
  String location =
"http://localhost:8080/graphlayout/service.xhtml?service=graph&graphname="
    + URLEncoder.~~encode~~(graphname)
    + "&version="
    + URLEncoder.~~encode~~("" + version);
  URL url = new URL(location);
  URL base = new URL("http://localhost:8080/");
  File imagedir = new File("./test-results/" + graphname + "/");
  if (!imagedir.exists())
   imagedir.mkdir();

  Applet myApplet = new DynamicGraphViewerApplet(base, url, layout,
    imagedir.getAbsolutePath());
  myApplet.init();
  frame.getContentPane().add(myApplet);
  frame.setSize(900, 700);
  frame.setVisible(true);
  myApplet.start();

  File file = new File(dir);
  File[] files = file.listFiles(new FileFilter() {
   public boolean accept(File f) {
    return f.isFile() && f.getName().endsWith(".dot");
   }
  });
  File outputfile = new File(diroutput + file.getName() + suffix + ".txt");
  BufferedWriter writer = new BufferedWriter(new FileWriter(outputfile));
  writer.write("Graph name \t size \t dynamic \t graphviz\n");
  for (int i = 0; i < files.length; i++) {
   String name = ApplicationUtil.getNameWithoutExtension(files[i].getName());
   String size = name.substring(name.lastIndexOf("-") + 1);
   String[] localgraphs = service.getGraphsnapshots(name);
   tokens = localgraphs[0].split(":");
   graphname = tokens[0];
   version = tokens[1];
   long start = System.currentTimeMillis();
   ((DynamicGraphViewerApplet) myApplet).reload(localgraphs[0]);
   ((DynamicGraphViewerApplet) myApplet).repaint();
   long end = System.currentTimeMillis();
   long taken1 = (end - start);
   writer.write(name + "\t" + size + "\t" + taken1 + "\t" + "\n");
   System.out.println(name + "\t" + size + "\t" + taken1 + "\t" + "\n");
  }
  writer.flush();
  writer.close();

  frame.dispose();
 }
```

Main function of the overall performance test program

```
public void testAll(String dir, String diroutput, String suffix)
   throws Exception {
  String[] graphs = service.getAllGraphVersions();
  StringBuffer buffer = new StringBuffer();
  String first = graphs[0];
  for (int i = 0; i < graphs.length; i++) {
   buffer.append(graphs[i] + ",");
  }
  String[] tokens = first.split(":");
  String graphname = tokens[0];
  String version = tokens[1];

  JFrame frame = new JFrame("graph viewer");
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  String layout = buffer.toString();
  String location =
"http://localhost:8080/graphlayout/service.xhtml?service=graph&graphname="
    + URLEncoder.~~encode~~(graphname)
    + "&version="
    + URLEncoder.~~encode~~("" + version);
  URL url = new URL(location);
  URL base = new URL("http://localhost:8080/");
  File imagedir = new File("./test-results/" + graphname + "/");
  if (!imagedir.exists())
   imagedir.mkdir();

  Applet myApplet = new DynamicGraphViewerApplet(base, url, layout,
    imagedir.getAbsolutePath());
  myApplet.init();
  frame.getContentPane().add(myApplet);
  frame.setSize(900, 700);
  frame.setVisible(true);
  myApplet.start();

  File file = new File(dir);
  File[] files = file.listFiles(new FileFilter() {
   public boolean accept(File f) {
    return f.isFile() && f.getName().endsWith(".dot");
   }
  });
  File outputfile = new File(diroutput + file.getName() + suffix + ".txt");
  BufferedWriter writer = new BufferedWriter(new FileWriter(outputfile));
  writer.write("Graph name \t size \t dynamic \t graphviz\n");
  for (int i = 0; i < files.length; i++) {
   String name = ApplicationUtil.getNameWithoutExtension(files[i].getName());
   delete(name);
   long taken1 = testConstrainedFrameworkPerformance(files[i]);

   String size = name.substring(name.lastIndexOf("-") + 1);
   String[] localgraphs = service.getGraphsnapshots(name);
   tokens = localgraphs[0].split(":");
   graphname = tokens[0];
   version = tokens[1];
   long start = System.currentTimeMillis();
   ((DynamicGraphViewerApplet) myApplet).reload(localgraphs[0]);
   ((DynamicGraphViewerApplet) myApplet).repaint();
   long end = System.currentTimeMillis();
   long taken2 = (end - start);
   long total = taken1 + taken2;

   writer.write(name + "\t" + size + "\t" + total + "\t" + "\n");
   System.out.println(name + "\t" + size + "\t" + total + "\t" + "\n");
  }
  writer.flush();
  writer.close();

  frame.dispose();
 }
```

Main function of the test program that measures the I/O cost of data retrieval due to dynamic operation

```
public void testAll() throws Exception {
  MockGraphAction action = new MockGraphAction(1);
  File[] dirs = { new File("./testfiles/real-dataset/small-data-set"),
    new File("./testfiles/real-dataset/medium-data-set"),
    new File("./testfiles/real-dataset/large-data-set"), };
  for (int m = 0; m < dirs.length; m++) {
   File[] files = dirs[m].listFiles(new FileFilter() {
    @Override public boolean accept(File pathname) {
     // TODO Auto-generated method stub
     return pathname.getName().endsWith("dynamic.txt");
    }
   });
   // initialize
   for (int j = 0; j < files.length; j++) {
    init(files[j], action);
   }
   File outputfile = new File(resultfolder, dirs[m].getName()
     + "-retrieval-cost-dynamic.txt");
   BufferedWriter writer = new BufferedWriter(new FileWriter(outputfile));
   writer.write("graph name\tsize\toperation\truntime\n");

   for (int i = 0; i < 6; i++) { // six operations
    for (int j = 0; j < files.length; j++) {
     callTest(files[j], writer, action, i);
    }
   }

   writer.flush();
   writer.close();
  }
 }

 public void delete(String graphName) throws Exception {
  general.update("delete from vertex where graph_name=?", graphName);
  general.update("delete from vertex_snapshot where graph_name=?", graphName);
  general.update("delete from order_constraint_snapshot where graph_name=?",
    graphName);
  general.update("delete from order_constraint where graph_name=?", graphName);
  general.update("delete from edge where graph_name=?", graphName);
  general.update("delete from edge_snapshot where graph_name=?", graphName);
  general.update("delete from layer where graph_name=?", graphName);
  general.update("delete from layer_snapshot where graph_name=?", graphName);
  general.update("delete from layout_snapshot where name=?", graphName);
  general.update("delete from layout where name=?", graphName);

 }

 public void callTest(File testfile, BufferedWriter writer,
   MockGraphAction action, int active) throws Exception {
  SimpleGraph graph = null;

  String name = ApplicationUtil.getNameWithoutExtension(testfile.getName());
  name = name.substring(0, name.lastIndexOf("-"));

  BufferedReader reader = new BufferedReader(new FileReader(testfile));

  String line = null;
  int count = 0;
  while (null != (line = reader.readLine())) {
   if (count == active) {
    System.out.println("testing ...." + line);

    Object result = action.execute(line, writer);
    graph = (SimpleGraph) result;

   }
   count++;
  }
  reader.close();

 }
```

Main function of the test program that measures the I/O cost of data saving due to dynamic operation

```
public void testAll() throws Exception {
  MockGraphAction2 action = new MockGraphAction2(1);
  File[] dirs = { new File("./testfiles/real-dataset/small-data-set"),
    new File("./testfiles/real-dataset/medium-data-set"),
    new File("./testfiles/real-dataset/large-data-set"), };
  for (int m = 0; m < dirs.length; m++) {
   File[] files = dirs[m].listFiles(new FileFilter() {
    @Override public boolean accept(File pathname) {
     // TODO Auto-generated method stub
     return pathname.getName().endsWith("dynamic.txt");
    }
   });
   // initialize
   for (int j = 0; j < files.length; j++) {
    init(files[j], action);
   }
   File outputfile = new File(resultfolder, dirs[m].getName()
     + "-savedb-cost-dynamic.txt");
   BufferedWriter writer = new BufferedWriter(new FileWriter(outputfile));
   writer.write("graph name\tsize\toperation\truntime\n");

   for (int i = 0; i < 6; i++) { // six operations
    for (int j = 0; j < files.length; j++) {
     callTest(files[j], writer, action, i);
    }
   }

   writer.flush();
   writer.close();
  }
 }

 public void delete(String graphName) throws Exception {
  general.update("delete from vertex where graph_name=?", graphName);
  general.update("delete from vertex_snapshot where graph_name=?", graphName);
  general.update("delete from order_constraint_snapshot where graph_name=?",
    graphName);
  general.update("delete from order_constraint where graph_name=?", graphName);
  general.update("delete from edge where graph_name=?", graphName);
  general.update("delete from edge_snapshot where graph_name=?", graphName);
  general.update("delete from layer where graph_name=?", graphName);
  general.update("delete from layer_snapshot where graph_name=?", graphName);
  general.update("delete from layout_snapshot where name=?", graphName);
  general.update("delete from layout where name=?", graphName);

 }

 public void callTest(File testfile, BufferedWriter writer,
   MockGraphAction2 action, int active) throws Exception {
```

```
  SimpleGraph graph = null;

  String name = ApplicationUtil.getNameWithoutExtension(testfile.getName());
  name = name.substring(0, name.lastIndexOf("-"));

  BufferedReader reader = new BufferedReader(new FileReader(testfile));

  String line = null;
  int count = 0;
  while (null != (line = reader.readLine())) {
   if (count == active) {
    System.out.println("testing ...." + line);

    Object result = action.execute(line, writer);
    graph = (SimpleGraph) result;

   }
   count++;
  }
  reader.close();

 }
```

Main function of the dynamic performance test program

```
public void test4() throws Exception {
  GraphAction action = new GraphAction(1);
  File[] dirs = { new File("./testfiles/real-dataset/small-data-set"),
    new File("./testfiles/real-dataset/medium-data-set"),
    new File("./testfiles/real-dataset/large-data-set"), };
  for (int m = 0; m < dirs.length; m++) {
   File[] files = dirs[m].listFiles(new FileFilter() {
    @Override public boolean accept(File pathname) {
     // TODO Auto-generated method stub
     return pathname.getName().endsWith("dynamic.txt");
    }
   });
   // initialize
   for (int j = 0; j < files.length; j++) {
    init(files[j], action);
   }
   File outputfile = new File(resultfolder, dirs[m].getName()
     + "-dynamic-3.txt");
   BufferedWriter writer = new BufferedWriter(new FileWriter(outputfile));
   writer.write("graph name\tsize\toperation\truntime\n");

   for (int i = 0; i < 6; i++) {
    for (int j = 0; j < files.length; j++) {
     test2(files[j], writer, action, i);
    }
   }

   writer.flush();
   writer.close();
  }
 }
 public void test(File testfile, BufferedWriter writer) throws Exception {
  GraphAction action = new GraphAction(1);
  SimpleGraph graph = null;

  String name = ApplicationUtil.getNameWithoutExtension(testfile.getName());
  name = name.substring(0, name.lastIndexOf("-"));

  String size = name.substring(name.lastIndexOf("-") + 1);
  delete(name);
  String origname = name + ".dot";
  File orig = new File(testfile.getParentFile(), origname);
  BufferedReader reader = new BufferedReader(new FileReader(orig));

  graph = (SimpleGraph) action.execute(reader);
  reader.close();

  reader = new BufferedReader(new FileReader(testfile));

  String line = null;
  while (null != (line = reader.readLine())) {
   System.out.println("testing ...." + line);
   long start = System.currentTimeMillis();
   Object result = action.execute(line);
   graph = (SimpleGraph) result;
   long end = System.currentTimeMillis();
   System.out.println(name + "\t" + (end - start));
   String op = line.substring(0, line.indexOf("\""));
   op = op.toLowerCase();
   writer.write(size + "\t" + op + "\t" + (end - start) + "\n");
  }
  reader.close();

 }
```

Main function of the Graphviz performance test program

```
public void testSmallDataSet() throws Exception {
  System.out.println("test small dataset ");
  testMany(testfolder + "small-data-set");
 }

 public void testMediumDataSet() throws Exception {
  System.out.println("test medium dataset ");
  testMany(testfolder + "medium-data-set");
 }

 public void testLargeDataSet() throws Exception {
  System.out.println("test large dataset ");
  testMany(testfolder + "large-data-set");
 }

 private void testMany(String dir) throws Exception {
  File file = new File(dir);
  File outputdir = new File(resultfolder, file.getName());
  File outputfile = new File(outputdir, "graphviz-result.txt");
  BufferedWriter writer = new BufferedWriter(new FileWriter(outputfile));
  File[] files = file.listFiles(new FileFilter() {
   public boolean accept(File f) {
    return f.isFile() && f.getName().endsWith(".dot");
   }
  });
  for (int i = 0; i < files.length; i++) {
   runTest(files[i], writer);
  }
  writer.flush();
  writer.close();
 }

 private void runTest(File testfile, BufferedWriter writer) throws Exception {
  long start = System.currentTimeMillis();
  String name = ApplicationUtil.getNameWithoutExtension(testfile.getName());
  File outputdir = new File(resultfolder, testfile.getParentFile().getName());
  if (!outputdir.exists())
   outputdir.mkdir();
  File outputfile = new File(outputdir, name + ".jpg");
  // System.out.println(outputfile.getAbsolutePath());
  ProcessBuilder pb = new ProcessBuilder("dot", "-Tjpeg", "-o"
    + outputfile.getAbsolutePath(), testfile.getAbsolutePath());
  Process p = pb.start();
```

```
   int i = p.waitFor();
   long end = System.currentTimeMillis();
   System.out.println(testfile.getName() + "\t" + (end - start));
   writer.write(testfile.getName() + "\t" + (end - start) + "\n");

  }
```

Appendix C

Performance Test Results for Dynamic Operations

Test result: I/O cost of data retrieval due to the Add edges operation

Graph name	Size	Operation	Run time (milliseconds)
org-parent-child-conversion-263	263	Add edges	15
org-parent-child-conversion-265	265	Add edges	15
org-parent-child-conversion-276	276	Add edges	16
org-parent-child-conversion-277	277	Add edges	16
org-parent-child-conversion-306	306	Add edges	31
org-parent-child-conversion-309	309	Add edges	16
process-parent-child-conversion-332	332	Add edges	31
process-parent-child-conversion-337	337	Add edges	15
org-parent-child-conversion-351	351	Add edges	31
process-parent-child-conversion-357	357	Add edges	16
org-parent-child-conversion-361	361	Add edges	16
org-parent-child-conversion-386	386	Add edges	31
process-parent-child-conversion-422	422	Add edges	32
org-parent-child-conversion-460	460	Add edges	31
process-parent-child-conversion-526	526	Add edges	31
org-parent-child-conversion-735	735	Add edges	15
org-parent-child-conversion-807	807	Add edges	31
org-parent-child-conversion-856	856	Add edges	31
org-parent-child-conversion-888	888	Add edges	31
org-parent-child-conversion-1063	1063	Add edges	47
process-parent-child-conversion-1112	1112	Add edges	31
org-parent-child-conversion-1444	1444	Add edges	63
org-parent-child-conversion-1733	1733	Add edges	63
process-parent-child-conversion-1849	1849	Add edges	78
org-parent-child-conversion-4164	4164	Add edges	125
process-parent-child-conversion-4495	4495	Add edges	94

Test result: I/O cost of data saving due to the Add edges operation

Graph name	Size	Operation	Run time (milliseconds)
org-parent-child-conversion-263	263	Add edges	94
org-parent-child-conversion-265	265	Add edges	63
org-parent-child-conversion-276	276	Add edges	62
org-parent-child-conversion-277	277	Add edges	62
org-parent-child-conversion-306	306	Add edges	94
org-parent-child-conversion-309	309	Add edges	109
process-parent-child-conversion-332	332	Add edges	125
process-parent-child-conversion-337	337	Add edges	109
org-parent-child-conversion-351	351	Add edges	110
process-parent-child-conversion-357	357	Add edges	140
org-parent-child-conversion-361	361	Add edges	63
org-parent-child-conversion-386	386	Add edges	63
process-parent-child-conversion-422	422	Add edges	141
org-parent-child-conversion-460	460	Add edges	94
process-parent-child-conversion-526	526	Add edges	172
org-parent-child-conversion-735	735	Add edges	93
org-parent-child-conversion-807	807	Add edges	94
org-parent-child-conversion-856	856	Add edges	125
org-parent-child-conversion-888	888	Add edges	125
org-parent-child-conversion-1063	1063	Add edges	203
process-parent-child-conversion-1112	1112	Add edges	172
org-parent-child-conversion-1444	1444	Add edges	281
org-parent-child-conversion-1733	1733	Add edges	297
process-parent-child-conversion-1849	1849	Add edges	422
org-parent-child-conversion-4164	4164	Add edges	484
process-parent-child-conversion-4495	4495	Add edges	516

Test result: Total run time of the Add edges operation

Graph name	Size	Operation	Run time (milliseconds)
org-parent-child-conversion-263	263	Add edges	124
org-parent-child-conversion-265	265	Add edges	86
org-parent-child-conversion-276	276	Add edges	87
org-parent-child-conversion-277	277	Add edges	103
org-parent-child-conversion-306	306	Add edges	165
org-parent-child-conversion-309	309	Add edges	145
process-parent-child-conversion-332	332	Add edges	146
process-parent-child-conversion-337	337	Add edges	138
org-parent-child-conversion-351	351	Add edges	155
process-parent-child-conversion-357	357	Add edges	193
org-parent-child-conversion-361	361	Add edges	83
org-parent-child-conversion-386	386	Add edges	137
process-parent-child-conversion-422	422	Add edges	178
org-parent-child-conversion-460	460	Add edges	136
process-parent-child-conversion-526	526	Add edges	170
org-parent-child-conversion-735	735	Add edges	164
org-parent-child-conversion-807	807	Add edges	176
org-parent-child-conversion-856	856	Add edges	155
org-parent-child-conversion-888	888	Add edges	187
org-parent-child-conversion-1063	1063	Add edges	179
process-parent-child-conversion-1112	1112	Add edges	176
org-parent-child-conversion-1444	1444	Add edges	265
org-parent-child-conversion-1733	1733	Add edges	312
process-parent-child-conversion-1849	1849	Add edges	820
org-parent-child-conversion-4164	4164	Add edges	558
process-parent-child-conversion-4495	4495	Add edges	599

Test result: I/O cost of data retrieval due to the Add vertices operation

Graph name	Size	Operation	Run time (milliseconds)
org-parent-child-conversion-263	263	Add vertices	32
org-parent-child-conversion-265	265	Add vertices	15
org-parent-child-conversion-276	276	Add vertices	15
org-parent-child-conversion-277	277	Add vertices	16
org-parent-child-conversion-306	306	Add vertices	16
org-parent-child-conversion-309	309	Add vertices	16
process-parent-child-conversion-332	332	Add vertices	16
process-parent-child-conversion-337	337	Add vertices	16
org-parent-child-conversion-351	351	Add vertices	15
process-parent-child-conversion-357	357	Add vertices	16
org-parent-child-conversion-361	361	Add vertices	15
org-parent-child-conversion-386	386	Add vertices	15
process-parent-child-conversion-422	422	Add vertices	15
org-parent-child-conversion-460	460	Add vertices	15
process-parent-child-conversion-526	526	Add vertices	16
org-parent-child-conversion-735	735	Add vertices	16
org-parent-child-conversion-807	807	Add vertices	31
org-parent-child-conversion-856	856	Add vertices	15
org-parent-child-conversion-888	888	Add vertices	16
org-parent-child-conversion-1063	1063	Add vertices	15
process-parent-child-conversion-1112	1112	Add vertices	32
org-parent-child-conversion-1444	1444	Add vertices	31
org-parent-child-conversion-1733	1733	Add vertices	47
process-parent-child-conversion-1849	1849	Add vertices	62
org-parent-child-conversion-4164	4164	Add vertices	94
process-parent-child-conversion-4495	4495	Add vertices	110

Test result: I/O cost of data saving due to the Add vertices operation

Graph name	Size	Operation	Run time (milliseconds)
org-parent-child-conversion-263	263	Add vertices	31
org-parent-child-conversion-265	265	Add vertices	47
org-parent-child-conversion-276	276	Add vertices	31
org-parent-child-conversion-277	277	Add vertices	47
org-parent-child-conversion-306	306	Add vertices	47
org-parent-child-conversion-309	309	Add vertices	46
process-parent-child-conversion-332	332	Add vertices	62
process-parent-child-conversion-337	337	Add vertices	78
org-parent-child-conversion-351	351	Add vertices	62
process-parent-child-conversion-357	357	Add vertices	94
org-parent-child-conversion-361	361	Add vertices	47
org-parent-child-conversion-386	386	Add vertices	47
process-parent-child-conversion-422	422	Add vertices	109
org-parent-child-conversion-460	460	Add vertices	62
process-parent-child-conversion-526	526	Add vertices	94
org-parent-child-conversion-735	735	Add vertices	63
org-parent-child-conversion-807	807	Add vertices	78
org-parent-child-conversion-856	856	Add vertices	94
org-parent-child-conversion-888	888	Add vertices	94
org-parent-child-conversion-1063	1063	Add vertices	188
process-parent-child-conversion-1112	1112	Add vertices	140
org-parent-child-conversion-1444	1444	Add vertices	188
org-parent-child-conversion-1733	1733	Add vertices	234
process-parent-child-conversion-1849	1849	Add vertices	296
org-parent-child-conversion-4164	4164	Add vertices	453
process-parent-child-conversion-4495	4495	Add vertices	407

Test result: Total run time of the Add vertices operation

Graph name	Size	Operation	Run time (milliseconds)
org-parent-child-conversion-263	263	Add vertices	89
org-parent-child-conversion-265	265	Add vertices	76
org-parent-child-conversion-276	276	Add vertices	72
org-parent-child-conversion-277	277	Add vertices	66
org-parent-child-conversion-306	306	Add vertices	95
org-parent-child-conversion-309	309	Add vertices	85
process-parent-child-conversion-332	332	Add vertices	90
process-parent-child-conversion-337	337	Add vertices	106
org-parent-child-conversion-351	351	Add vertices	91
process-parent-child-conversion-357	357	Add vertices	144
org-parent-child-conversion-361	361	Add vertices	72
org-parent-child-conversion-386	386	Add vertices	87
process-parent-child-conversion-422	422	Add vertices	143
org-parent-child-conversion-460	460	Add vertices	262
process-parent-child-conversion-526	526	Add vertices	111
org-parent-child-conversion-735	735	Add vertices	133
org-parent-child-conversion-807	807	Add vertices	131
org-parent-child-conversion-856	856	Add vertices	135
org-parent-child-conversion-888	888	Add vertices	131
org-parent-child-conversion-1063	1063	Add vertices	184
process-parent-child-conversion-1112	1112	Add vertices	171
org-parent-child-conversion-1444	1444	Add vertices	167
org-parent-child-conversion-1733	1733	Add vertices	193
process-parent-child-conversion-1849	1849	Add vertices	288
org-parent-child-conversion-4164	4164	Add vertices	512
process-parent-child-conversion-4495	4495	Add vertices	516

Test result: I/O cost of data retrieval due to the Remove vertices operation

Graph name	Size	Operation	Run time (milliseconds)
org-parent-child-conversion-263	263	Remove vertices	16
org-parent-child-conversion-265	265	Remove vertices	16
org-parent-child-conversion-276	276	Remove vertices	16
org-parent-child-conversion-277	277	Remove vertices	15
org-parent-child-conversion-306	306	Remove vertices	31
org-parent-child-conversion-309	309	Remove vertices	16
process-parent-child-conversion-332	332	Remove vertices	15
process-parent-child-conversion-337	337	Remove vertices	31
org-parent-child-conversion-351	351	Remove vertices	15
process-parent-child-conversion-357	357	Remove vertices	16
org-parent-child-conversion-361	361	Remove vertices	94
org-parent-child-conversion-386	386	Remove vertices	32
process-parent-child-conversion-422	422	Remove vertices	31
org-parent-child-conversion-460	460	Remove vertices	31
process-parent-child-conversion-526	526	Remove vertices	31
org-parent-child-conversion-735	735	Remove vertices	47
org-parent-child-conversion-807	807	Remove vertices	46
org-parent-child-conversion-856	856	Remove vertices	47
org-parent-child-conversion-888	888	Remove vertices	62
org-parent-child-conversion-1063	1063	Remove vertices	32

process-parent-child-conversion-1112	1112	Remove vertices	47
org-parent-child-conversion-1444	1444	Remove vertices	94
org-parent-child-conversion-1733	1733	Remove vertices	47
process-parent-child-conversion-1849	1849	Remove vertices	78
org-parent-child-conversion-4164	4164	Remove vertices	406
process-parent-child-conversion-4495	4495	Remove vertices	188

Test result: I/O cost of data saving due to the Remove vertices operation

Graph name	Size	Operation	Run time (milliseconds)
org-parent-child-conversion-263	263	Remove vertices	93
org-parent-child-conversion-265	265	Remove vertices	125
org-parent-child-conversion-276	276	Remove vertices	62
org-parent-child-conversion-277	277	Remove vertices	93
org-parent-child-conversion-306	306	Remove vertices	78
org-parent-child-conversion-309	309	Remove vertices	109
process-parent-child-conversion-332	332	Remove vertices	109
process-parent-child-conversion-337	337	Remove vertices	63
org-parent-child-conversion-351	351	Remove vertices	110
process-parent-child-conversion-357	357	Remove vertices	78
org-parent-child-conversion-361	361	Remove vertices	109
org-parent-child-conversion-386	386	Remove vertices	109
process-parent-child-conversion-422	422	Remove vertices	109
org-parent-child-conversion-460	460	Remove vertices	125
process-parent-child-conversion-526	526	Remove vertices	125
org-parent-child-conversion-735	735	Remove vertices	156
org-parent-child-conversion-807	807	Remove vertices	250
org-parent-child-conversion-856	856	Remove vertices	203
org-parent-child-conversion-888	888	Remove vertices	250
org-parent-child-conversion-1063	1063	Remove vertices	203

process-parent-child-conversion-1112	1112	Remove vertices	266
org-parent-child-conversion-1444	1444	Remove vertices	407
org-parent-child-conversion-1733	1733	Remove vertices	297
process-parent-child-conversion-1849	1849	Remove vertices	360
org-parent-child-conversion-4164	4164	Remove vertices	1109
process-parent-child-conversion-4495	4495	Remove vertices	750

Test result: Total run time of the Remove vertices operation

Graph name	Size	Operation	Run time (milliseconds)
org-parent-child-conversion-263	263	Remove vertices	132
org-parent-child-conversion-265	265	Remove vertices	91
org-parent-child-conversion-276	276	Remove vertices	86
org-parent-child-conversion-277	277	Remove vertices	133
org-parent-child-conversion-306	306	Remove vertices	145
org-parent-child-conversion-309	309	Remove vertices	149
process-parent-child-conversion-332	332	Remove vertices	169
process-parent-child-conversion-337	337	Remove vertices	114
org-parent-child-conversion-351	351	Remove vertices	135
process-parent-child-conversion-357	357	Remove vertices	162
org-parent-child-conversion-361	361	Remove vertices	156
org-parent-child-conversion-386	386	Remove vertices	165
process-parent-child-conversion-422	422	Remove vertices	201
org-parent-child-conversion-460	460	Remove vertices	174
process-parent-child-conversion-526	526	Remove vertices	172
org-parent-child-conversion-735	735	Remove vertices	241
org-parent-child-conversion-807	807	Remove vertices	356
org-parent-child-conversion-856	856	Remove vertices	274
org-parent-child-conversion-888	888	Remove vertices	345
org-parent-child-conversion-1063	1063	Remove vertices	181
process-parent-child-conversion-1112	1112	Remove vertices	305
org-parent-child-conversion-1444	1444	Remove vertices	423
org-parent-child-conversion-1733	1733	Remove vertices	283
process-parent-child-conversion-1849	1849	Remove vertices	397
org-parent-child-conversion-4164	4164	Remove vertices	1188
process-parent-child-conversion-4495	4495	Remove vertices	873

Test result: I/O cost of data retrieval due to the Remove edges operation

Graph name	Size	Operation	Run time (milliseconds)
org-parent-child-conversion-263	263	Remove edges	15
org-parent-child-conversion-265	265	Remove edges	16
org-parent-child-conversion-276	276	Remove edges	31
org-parent-child-conversion-277	277	Remove edges	15
org-parent-child-conversion-306	306	Remove edges	16
org-parent-child-conversion-309	309	Remove edges	16
process-parent-child-conversion-332	332	Remove edges	16
process-parent-child-conversion-337	337	Remove edges	32
org-parent-child-conversion-351	351	Remove edges	15
process-parent-child-conversion-357	357	Remove edges	15
org-parent-child-conversion-361	361	Remove edges	31
org-parent-child-conversion-386	386	Remove edges	32
process-parent-child-conversion-422	422	Remove edges	31
org-parent-child-conversion-460	460	Remove edges	31
process-parent-child-conversion-526	526	Remove edges	47
org-parent-child-conversion-735	735	Remove edges	47
org-parent-child-conversion-807	807	Remove edges	62
org-parent-child-conversion-856	856	Remove edges	78
org-parent-child-conversion-888	888	Remove edges	47
org-parent-child-conversion-1063	1063	Remove edges	47
process-parent-child-conversion-1112	1112	Remove edges	31
org-parent-child-conversion-1444	1444	Remove edges	63
org-parent-child-conversion-1733	1733	Remove edges	203
process-parent-child-conversion-1849	1849	Remove edges	78
org-parent-child-conversion-4164	4164	Remove edges	969
process-parent-child-conversion-4495	4495	Remove edges	156

Test result: I/O cost of data saving due to the Remove edges operation

Graph name	Size	Operation	Run time (milliseconds)
org-parent-child-conversion-263	263	Remove edges	94
org-parent-child-conversion-265	265	Remove edges	109
org-parent-child-conversion-276	276	Remove edges	94
org-parent-child-conversion-277	277	Remove edges	62
org-parent-child-conversion-306	306	Remove edges	78
org-parent-child-conversion-309	309	Remove edges	109
process-parent-child-conversion-332	332	Remove edges	109
process-parent-child-conversion-337	337	Remove edges	109
org-parent-child-conversion-351	351	Remove edges	110
process-parent-child-conversion-357	357	Remove edges	125
org-parent-child-conversion-361	361	Remove edges	125
org-parent-child-conversion-386	386	Remove edges	125
process-parent-child-conversion-422	422	Remove edges	141
org-parent-child-conversion-460	460	Remove edges	141
process-parent-child-conversion-526	526	Remove edges	203
org-parent-child-conversion-735	735	Remove edges	172
org-parent-child-conversion-807	807	Remove edges	266
org-parent-child-conversion-856	856	Remove edges	281
org-parent-child-conversion-888	888	Remove edges	141
org-parent-child-conversion-1063	1063	Remove edges	266
process-parent-child-conversion-1112	1112	Remove edges	172
org-parent-child-conversion-1444	1444	Remove edges	266
org-parent-child-conversion-1733	1733	Remove edges	671
process-parent-child-conversion-1849	1849	Remove edges	375
org-parent-child-conversion-4164	4164	Remove edges	1594
process-parent-child-conversion-4495	4495	Remove edges	672

Test result: Remove edges operation

Graph name	Size	Operation	Run time (milliseconds)
org-parent-child-conversion-263	263	Remove edges	144
org-parent-child-conversion-265	265	Remove edges	142
org-parent-child-conversion-276	276	Remove edges	122
org-parent-child-conversion-277	277	Remove edges	104
org-parent-child-conversion-306	306	Remove edges	151
org-parent-child-conversion-309	309	Remove edges	166
process-parent-child-conversion-332	332	Remove edges	140
process-parent-child-conversion-337	337	Remove edges	155
org-parent-child-conversion-351	351	Remove edges	147
process-parent-child-conversion-357	357	Remove edges	150
org-parent-child-conversion-361	361	Remove edges	160
org-parent-child-conversion-386	386	Remove edges	208
process-parent-child-conversion-422	422	Remove edges	201
org-parent-child-conversion-460	460	Remove edges	220
process-parent-child-conversion-526	526	Remove edges	242
org-parent-child-conversion-735	735	Remove edges	274
org-parent-child-conversion-807	807	Remove edges	402
org-parent-child-conversion-856	856	Remove edges	352
org-parent-child-conversion-888	888	Remove edges	214
org-parent-child-conversion-1063	1063	Remove edges	289
process-parent-child-conversion-1112	1112	Remove edges	221
org-parent-child-conversion-1444	1444	Remove edges	288
org-parent-child-conversion-1733	1733	Remove edges	759
process-parent-child-conversion-1849	1849	Remove edges	442
org-parent-child-conversion-4164	4164	Remove edges	2048
process-parent-child-conversion-4495	4495	Remove edges	864

Test result: I/O cost of data retrieval due to the Set order constraints operation

Graph name	Size	Operation	Run time (milliseconds)
org-parent-child-conversion-306	306	Set order	16
org-parent-child-conversion-386	386	Set order	15
org-parent-child-conversion-460	460	Set order	16
org-parent-child-conversion-735	735	Set order	15
org-parent-child-conversion-888	888	Set order	16
org-parent-child-conversion-1063	1063	Set order	31

Test result: I/O cost of data saving due to the Set order constraints operation

Graph name	Size	Operation	Run time (milliseconds)
org-parent-child-conversion-263	263	Set order	16
org-parent-child-conversion-265	265	Set order	15
org-parent-child-conversion-276	276	Set order	15
org-parent-child-conversion-277	277	Set order	16
org-parent-child-conversion-306	306	Set order	47
org-parent-child-conversion-309	309	Set order	16
process-parent-child-conversion-332	332	Set order	15
process-parent-child-conversion-337	337	Set order	0
org-parent-child-conversion-351	351	Set order	15
process-parent-child-conversion-357	357	Set order	0
org-parent-child-conversion-361	361	Set order	0
org-parent-child-conversion-386	386	Set order	47
process-parent-child-conversion-422	422	Set order	15
org-parent-child-conversion-460	460	Set order	62
process-parent-child-conversion-526	526	Set order	15
org-parent-child-conversion-735	735	Set order	79
org-parent-child-conversion-807	807	Set order	0
org-parent-child-conversion-856	856	Set order	0
org-parent-child-conversion-888	888	Set order	94
org-parent-child-conversion-1063	1063	Set order	156
process-parent-child-conversion-1112	1112	Set order	15
org-parent-child-conversion-1444	1444	Set order	16
org-parent-child-conversion-1733	1733	Set order	16
process-parent-child-conversion-1849	1849	Set order	16
org-parent-child-conversion-4164	4164	Set order	16
process-parent-child-conversion-4495	4495	Set order	16

Test result: Set order constraints operation

Graph name	Size	Operation	Run time (milliseconds)
org-parent-child-conversion-263	263	Set order	45
org-parent-child-conversion-265	265	Set order	29
org-parent-child-conversion-276	276	Set order	27
org-parent-child-conversion-277	277	Set order	26
org-parent-child-conversion-306	306	Set order	107
org-parent-child-conversion-309	309	Set order	24
process-parent-child-conversion-332	332	Set order	28
process-parent-child-conversion-337	337	Set order	24
org-parent-child-conversion-351	351	Set order	38
process-parent-child-conversion-357	357	Set order	34
org-parent-child-conversion-361	361	Set order	22
org-parent-child-conversion-386	386	Set order	158
process-parent-child-conversion-422	422	Set order	22
org-parent-child-conversion-460	460	Set order	117
process-parent-child-conversion-526	526	Set order	22
org-parent-child-conversion-735	735	Set order	136
org-parent-child-conversion-807	807	Set order	20
org-parent-child-conversion-856	856	Set order	48
org-parent-child-conversion-888	888	Set order	161
org-parent-child-conversion-1063	1063	Set order	190
process-parent-child-conversion-1112	1112	Set order	22
org-parent-child-conversion-1444	1444	Set order	25
org-parent-child-conversion-1733	1733	Set order	29
process-parent-child-conversion-1849	1849	Set order	47
org-parent-child-conversion-4164	4164	Set order	31
process-parent-child-conversion-4495	4495	Set order	28

Test result: I/O cost of data retrieval due to the Drop order constraints operation

Graph name	Size	Operation	Run time (milliseconds)
org-parent-child-conversion-263	263	Drop order	16
org-parent-child-conversion-265	265	Drop order	16
org-parent-child-conversion-276	276	Drop order	16
org-parent-child-conversion-277	277	Drop order	16
org-parent-child-conversion-306	306	Drop order	16
org-parent-child-conversion-309	309	Drop order	15
process-parent-child-conversion-332	332	Drop order	16
process-parent-child-conversion-337	337	Drop order	16
org-parent-child-conversion-351	351	Drop order	15
process-parent-child-conversion-357	357	Drop order	15
org-parent-child-conversion-361	361	Drop order	16
org-parent-child-conversion-386	386	Drop order	16
process-parent-child-conversion-422	422	Drop order	31
org-parent-child-conversion-460	460	Drop order	0
process-parent-child-conversion-526	526	Drop order	140
org-parent-child-conversion-735	735	Drop order	31
org-parent-child-conversion-807	807	Drop order	31
org-parent-child-conversion-856	856	Drop order	16
org-parent-child-conversion-888	888	Drop order	16
org-parent-child-conversion-1063	1063	Drop order	15
process-parent-child-conversion-1112	1112	Drop order	31
org-parent-child-conversion-1444	1444	Drop order	47
org-parent-child-conversion-1733	1733	Drop order	47
process-parent-child-conversion-1849	1849	Drop order	62
org-parent-child-conversion-4164	4164	Drop order	109
process-parent-child-conversion-4495	4495	Drop order	93

Test result: I/O cost of data saving due to the Drop order constraints operation

Graph name	Size	Operation	Run time (milliseconds)
org-parent-child-conversion-263	263	Drop order	31
org-parent-child-conversion-265	265	Drop order	47
org-parent-child-conversion-276	276	Drop order	31
org-parent-child-conversion-277	277	Drop order	31
org-parent-child-conversion-306	306	Drop order	47
org-parent-child-conversion-309	309	Drop order	47
process-parent-child-conversion-332	332	Drop order	47
process-parent-child-conversion-337	337	Drop order	47
org-parent-child-conversion-351	351	Drop order	62
process-parent-child-conversion-357	357	Drop order	63
org-parent-child-conversion-361	361	Drop order	63
org-parent-child-conversion-386	386	Drop order	47
process-parent-child-conversion-422	422	Drop order	78
org-parent-child-conversion-460	460	Drop order	62
process-parent-child-conversion-526	526	Drop order	63
org-parent-child-conversion-735	735	Drop order	94
org-parent-child-conversion-807	807	Drop order	94
org-parent-child-conversion-856	856	Drop order	94
org-parent-child-conversion-888	888	Drop order	110
org-parent-child-conversion-1063	1063	Drop order	156
process-parent-child-conversion-1112	1112	Drop order	109
org-parent-child-conversion-1444	1444	Drop order	141
org-parent-child-conversion-1733	1733	Drop order	172
process-parent-child-conversion-1849	1849	Drop order	204
org-parent-child-conversion-4164	4164	Drop order	422
process-parent-child-conversion-4495	4495	Drop order	344

Test result: Drop order constraints operation

Graph name	Size	Operation	Run time (milliseconds)
org-parent-child-conversion-263	263	Drop order	84
org-parent-child-conversion-265	265	Drop order	74
org-parent-child-conversion-276	276	Drop order	68
org-parent-child-conversion-277	277	Drop order	84
org-parent-child-conversion-306	306	Drop order	92
org-parent-child-conversion-309	309	Drop order	91
process-parent-child-conversion-332	332	Drop order	96
process-parent-child-conversion-337	337	Drop order	118
org-parent-child-conversion-351	351	Drop order	116
process-parent-child-conversion-357	357	Drop order	129
org-parent-child-conversion-361	361	Drop order	118
org-parent-child-conversion-386	386	Drop order	88
process-parent-child-conversion-422	422	Drop order	157
org-parent-child-conversion-460	460	Drop order	124
process-parent-child-conversion-526	526	Drop order	210
org-parent-child-conversion-735	735	Drop order	137
org-parent-child-conversion-807	807	Drop order	167
org-parent-child-conversion-856	856	Drop order	145
org-parent-child-conversion-888	888	Drop order	152
org-parent-child-conversion-1063	1063	Drop order	221
process-parent-child-conversion-1112	1112	Drop order	188
org-parent-child-conversion-1444	1444	Drop order	209
org-parent-child-conversion-1733	1733	Drop order	239
process-parent-child-conversion-1849	1849	Drop order	323
org-parent-child-conversion-4164	4164	Drop order	792
process-parent-child-conversion-4495	4495	Drop order	524

Appendix D

SQL Queries Used in Test Validation

Count the number of layers in a process graph layout:

```
select count(layer) from layer where graph_name='process-parent-child-conversion';
```

Count the number of vertices in a process graph layout:

```
select count(*) from vertex where graph_name='process-parent-child-conversion';
```

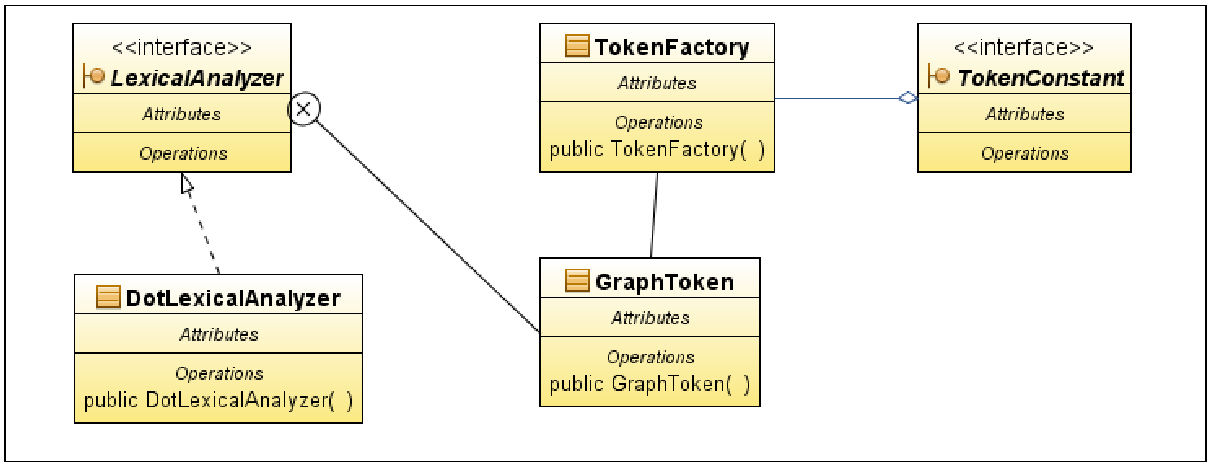
Find data generating dynamic operations:

```
select * from vertex where graph_name='org-parent-child-conversion' order by layer,
vertex_id;

select max(vertex_id) from vertex where graph_name='org-parent-child-conversion';

select * from edge where graph_name='org-parent-child-conversion' and head > 1;

select * from vertex where graph_name='org-parent-child-conversion' and layer=4 order by
position, vertex_id;
```

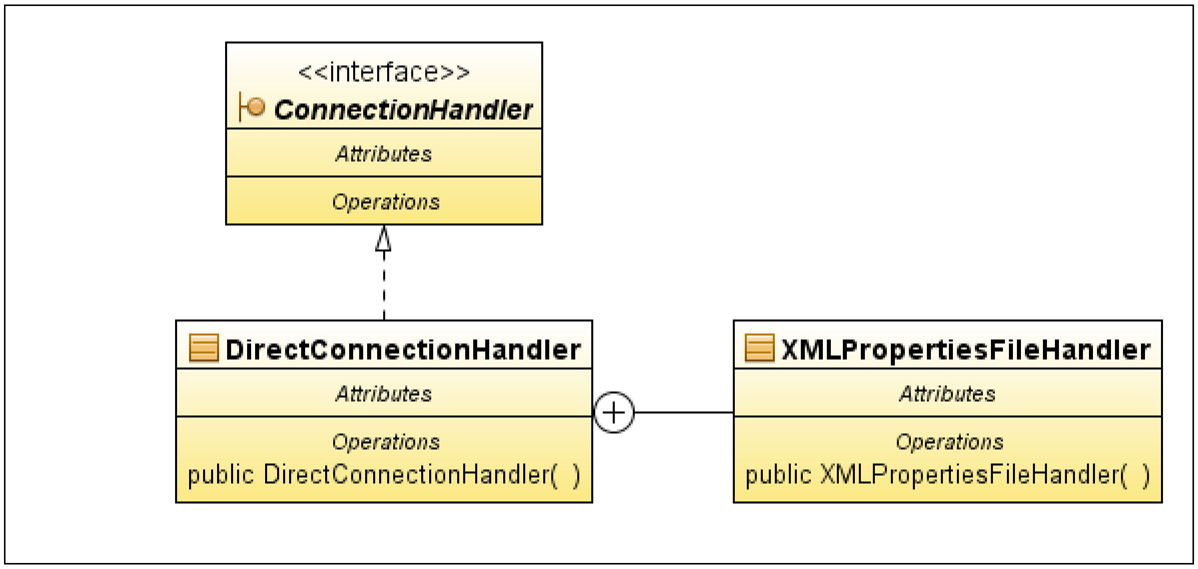
Appendix E

Class Diagrams for Helper Classes in the Constrained Graph Drawing Framework

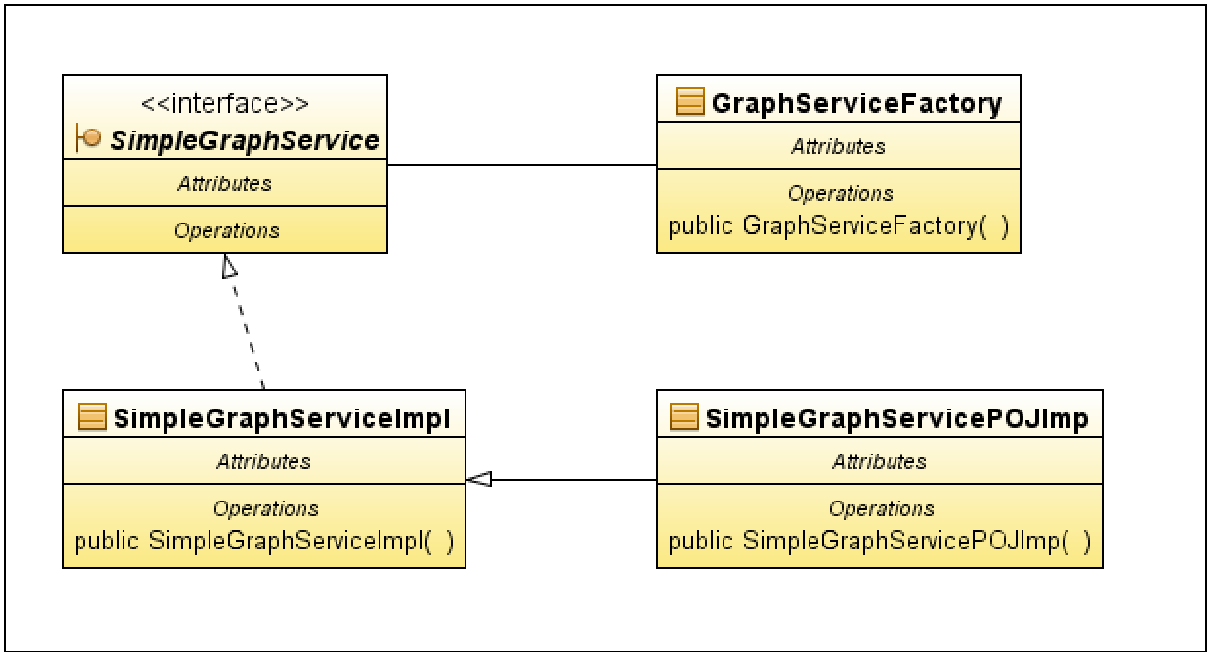
Lexical class diagram



Connection class diagram



Web service class diagram



Reference List

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

Berger, B., & Shor, P. W. (1990). Approximation algorithms for the maximum acyclic subgraph. *Proceedings of the First ACM-ISAM Symposium on Discrete Algorithms*, 236–243.

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 processors systems. *Acta Informica* 1, 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.

Davison, R., & Harell, D. (1996). Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics Vol. 15* (301–331)

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

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 combin., pp. 187–191. *Proceedings of the Australian Computer Science Conference*, 327–334.

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

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

Forster, M. (2004). A fast and simple heuristic for constrained two-layered crossing reduction. *Proc. Graph Drawing, GD 2004*, 206–216.

Frishman, Y., & Tal, A. (2007). On-line dynamic graph drawing. *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, 3,* 289–314. Boston: Kluwer.

Huang, W., & Eades, P. (2005). How people read graphs. *Asia Pacific Symposium on Information Visualization (APVIS 2005)*. Australia.

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. *Vol. 60 of Conferences in Research and Practice in Information Technology.*

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

Lin, X. Y. (1992). *Analysis of algorithms for drawing graphs*. PhD thesis, Department of Computer Science, 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. *Lecture Notes in Computer Science; Vol. 1731: Proceedings of the 7th international symposium on graph drawing* (pp. 217–224). London: Springer-Verlag.

Miriyala, K., & Tamassia, R. (1993). An incremental approach to aesthetic graph layout. *Computer-Aided Software Engineering*, *Proceeding of the Sixth International Workshop*, *Vol. 47. Iss 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*. Master's thesis, Florida State University, Tallahassee, Florida.

Rabani, Y. (2003). *Approximation Algorithms Lectures*. Retrieved May 02, 2007 from http://www.cs.technion.ac.il/~rabani/236521.04.wi.html

Raitner, M. (2004). Maintaining hierarchical graph views for dynamic graphs. *Technical Report, MIP-0403*, University of Passau, Passau, Germany.

Rudell, R. (1993). Dynamic variable ordering for ordered binary decision diagram. In *Proc. International Conference on Computer-Aided Design* (pp. 42–47).

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

Sander G. (1996). *Visualisierungstechniken f¨ur den Compilerbau.* PhD thesis, University of Saarbrücken.

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

Stedile, A. (2001). *JMFGraph - A modular framework for drawing graphs in Java*. 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. (2001). Graph layout for displaying data structures. In J. Marks, editor, Proc. GD'00, *volume 1984 of LNCS, pp. 241–252*. Springer, 2001.

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*. New Jersey: Prentice Hall.