University of Alberta

Library Release Form

Name of Author: Wenbin Ma

Title of Thesis: GDC: A Graph Drawing Application with Clustering Techniques

Degree: Master of Science

Year this Degree Granted: 2001

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Wenbin Ma
114 St - 89 Ave
Edmonton, Alberta
Canada, T6G 2C5

Date: ____________________

University of Alberta

GDC: A Graph Drawing Application with Clustering Techniques

by

Wenbin Ma

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2001

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **GDC: A Graph Drawing Application with Clustering Techniques** submitted by Wenbin Ma in partial fulfillment of the requirements for the degree of **Master of Science**.

Supervisor: Joe Culberson

Lorna Stewart

Andy Liu (Math Department)

Date:

Abstract

We investigate some common methods for three fundamental types of graph layout and attempt to characterize various methods with a sequence of heuristics according to a framework for each graph layout. We show the heuristics effect and their working principles in the drawing algorithms. By adding one preprocessing heuristic, we can improve some algorithms implemented in the existing libraries. We offer two efficient heuristics for counting the edge intersections under distinct layouts. Five drawing algorithms are also empirically studied in the thesis.

It has been shown that the problem of graph clustering is correlated with the problem of graph drawing. We investigate a classification of auxiliary clustering approaches for graph drawing. We also develop two clustering techniques and demonstrate that the clustering-based drawing algorithms can significantly improve the drawing aesthetic criteria for some types of graphs.

Acknowledgements

I would like express my deep gratitude to my supervisor, Joe Culberson, for his supervision, encouragement and motivation through the course of this work. Without his guidance, care and insightful comments, this work would not be possible.

I am grateful to my co-supervisor, Lorna Stewart, for her support and excellent course "Graph Theory". My thanks also goes to Stephen Wismath for technical discussions, Erkki Makinen for mailing me some of his interesting papers.

I would like to thank my classmates in the algorithm group, Yong Gao and Adam Beach. I would also like to acknowledge staff members and graduate students of the Department of Computing Science for providing such a friendly and enjoyable environment during my study here.

Finally, I am deeply indebted to my parents for their forever affection, patience and encouragement. I dedicate this thesis to my mother and father.

Wenbin Ma

January, 2001
Edmonton, Canada

Contents

List of Figures

List of Tables

Notations

General

area : the area of the drawing plane.

$d(u, v)$: the Euclidean distance between u and v.

$G = (V, E)$: the input graph for the drawing algorithm. V is the vertex set and E is the edge set.

FDP (Frozen Development Process): given the vertex number N of a graph and a random permutation S of all $\binom{N}{2}$ vertex pairs, an FDP is a series of graphs constructed by adding the edges between the vertex pair in sequence.

N : the number of vertices in an FDP.

n : the number of vertices of G.

v : an arbitrary vertex that belonging to V.

(x_v, y_v) is the coordinate of vertex v's position.

Section 2.3

$\delta(u, v)$: the desired Euclidean distance between u and v.

E**:** the global energy.

E_{uv} : the energy of v acted on by u.

f_{uv} : the spring force exerted on v by the *spring neighbor* u such that there is a *spring* between vertex u and v in the model.

$F(v)$: the force exerted on vertex v by the entire G.

g_{uv} : the electrical repulsion exerted on v by vertex u. It prevents v from being too close to u.

k : the spring stiffness parameter.

optl : the desired Euclidean length of an edge.

$s(u, v)$: the number of edges on the shortest path between u and v.

Section 2.4

k : the number of the hierarchy levels. $V = \{V_1, V_2, ..., V_k\}$and $V_i \bigcap V_j = \emptyset$ for $i \neq j$. .

$width$: the width of the drawing plane.

V_i : the subset of vertices on level i.

Chapter 3

$\overline{deg}$: the average degree of G.

$deg(v)$: the number of adjacent vertices of vertex v.

$N(v)$: the subset of vertices adjacent to v.

T : the largest index of the edges that cause vertices frozen same in an FDP.

Chapter 4

$Cluster(G)$: the number of clusters in graph G.

k : the number of clusters of G. This means that $V = \bigcup_{i=1}^{k} V_i$ and $V_i \bigcap V_j = \emptyset$ for $i \neq j$.

V_i : a cluster of G, i.e. a subset of V, $1 \leq i \leq k$.

$|V_i|$: the number of vertices in V_i.

$W(V_1, V_2, ..., V_k)$: the clustering criterion function that measures the correctness of all the k clusters of G.

Chapter 1

Introduction

Any domain that can be modeled as a collection of entities and pairwise relationships between these entities can be represented as a graph in which the entities are *vertices* and the relationships are *edges*. With the advent of high-resolution displays and high-speed computations, graph visualization has become more applicable in many fields, such as CASE tools, database systems, VLSI systems, and network systems design. The effectiveness of the visualization of a graph is dependent on how efficiently the associated diagram conveys information to the viewer. A *drawing* of a graph is an assignment of coordinates to each vertex and an assignment of routes to each edge. A good drawing is worth hundreds of words, but a poor drawing can be confusing and misleading. To automatically generate a good drawing for a graph, we need an algorithm that assigns a location for each vertex and a route for each edge; this is the *graph drawing algorithm*. G.D.Battista, P.Eades, R.Tamassia, and I.G.Tollis presented an annotated bibliography [BETT94] and a book [BETT99] on this area, and the problem is currently the subject of an annual "Graph Drawing Conference".

In this chapter, we provide a brief overview of the graph drawing problem in Section 1.1. The thesis is motivated by the desire to visualize the graph collapsing phenomenon associated with the 3-Coloring phase transition, which will be discussed in Section 1.2. We will introduce some well-known graph drawing libraries and editors as well as our application in Section 1.3.

In Chapter 2, we will provide an overview of graph layouts and some classic drawing methods. We will discuss some implementation issues concerning the properties of collapsed graphs and some drawing algorithms and evaluate the performance of drawing algorithms in Chapter 3. In Chapter 4, we will investigate graph clustering methods and present some clustering techniques to aid graph drawing algorithms. The system architecture as well as the implementation of some features of GDC are outlined in Chapter 5. Chapter 6 will provide a conclusion and ideas for future work.

1.1 Graph Drawing Overview

In this section, we will first introduce three fundamental parameters for graph drawing methodologies: conventions for drawing graphs, drawing aesthetic criteria and drawing constraints on graph drawings.

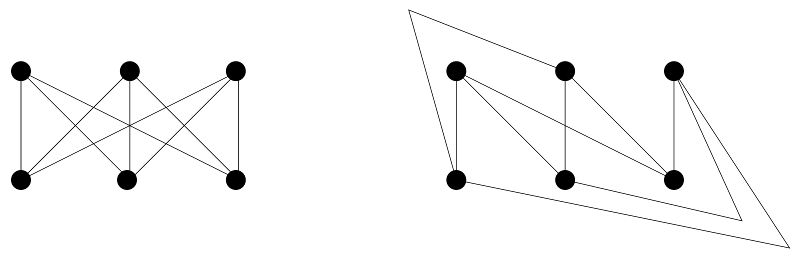
1.1.1 Drawing conventions

A basic rule that a drawing must satisfy to be admissible is called a *drawing convention*. Drawing conventions for graphs differ from one application area to another. Some of the most important conventions are listed below:

- Many graph drawing methods produce a *grid drawing*: the location of each vertex has integer coordinates.

- Figure 1.1(b) is a *polyline drawing*, where the curve representing each edge is a polyline, a chain of line segments. If each polyline is just a line segment, the drawing is a *straight-line drawing*, as seen in Figure 1.1 (a).

- In an *orthogonal drawing*, each edge is a polyline composed of straight line segments parallel to one of the coordinate axes. There is much research on orthogonal drawings because horizontal and vertical line segments are easy to follow. Figure 1.2 is an orthogonal drawing.



(a) A Straight-line drawing (b) A polyline drawing

Figure 1.1: Straight-line Drawing and Polyline Drawings of $K_{3,3}$

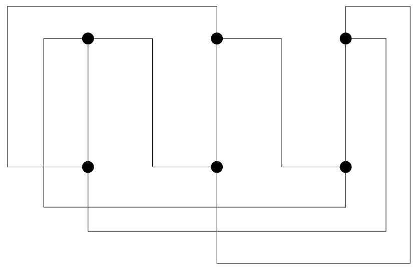


Figure 1.2: An orthogonal Drawing of $K_{3,3}$

1.1.2 Drawing Aesthetics

The primary requirement of graph drawing algorithms is that the output graph should be *readable*; that is, it should be easy to understand and follow. It is hard to model readability precisely because it varies from one application to another and from one human to another; these variations imply that there are many graph drawing problems according to the optimization goals which the algorithms try to achieve. These goals are called *drawing aesthetics*, some of which are as follows:

- Minimize the number of edge crossings. A graph drawing with none or few edge crossings is pleasing and easy to understand. The drawing in Figure 1.3(a) has many more edge crossings than that in Figure 1.3(b). Obviously, the drawing in Figure 1.3(b) is pleasing and easy to understand, but the drawing in Figure 1.3(a) is difficult to follow. Some recent work on crossings approximations can be found at [EGS2000].

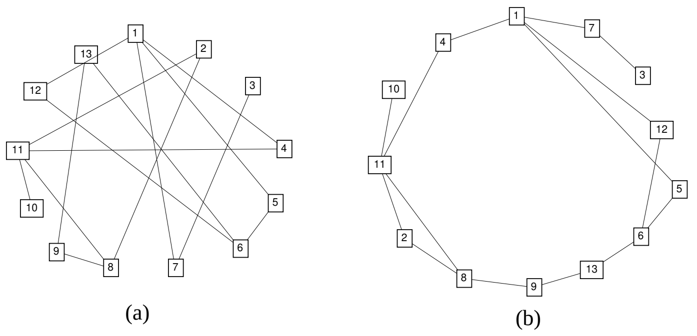


Figure 1.3: Two Drawings of the Same Graph

- Minimize the number of bends on the edges. In polyline drawings, the edge with fewer bends is easy to follow for many users [BETT99].

- Minimize the vertex distribution. The vertex distribution of a drawing is characterized by the Euclidean distance between every pair of vertices. This term prevents vertices from coming too close together.

- Maximize the symmetry display. Some mathematical models of symmetries in graph drawings are introduced in [HRM95][Man90].

These criteria, however, cannot be achieved optimally in polynomial time (unless $P = NP$) [Joh84]. Therefore, it is feasible to find near optimal solutions using heuristics. Furthermore, simultaneous optimization for several criteria might not be possible or lead to quality tradeoffs because there exist incompatible combinations [Joh84].

H.C.Purchase et al validated three graph drawing aesthetics: *maximization of symmetr*ies, *minimization of edge crossings*, and *minimization of bends* by empirically studying the human understanding of graphs [HRM95]. Their results indicate that minimizing edge bends and minimizing edge crossings are more helpful to human understanding than maximizing symmetry.

1.1.3 Constraints

Some constraints on graph drawing algorithms are [BETT99]:

- Cluster: A given subset of vertices should be placed close together.

- Shape: A given subgraph should be drawn with a predefined "shape".

The requirements of graph drawing algorithms can be modeled in terms of drawing conventions, drawing aesthetic criteria and drawing constraints. We will discuss graph drawing algorithms in Chapter 2.

1.2 Vertex Collapse Phenomenon in the Phase Transition of Graph Coloring

Random graphs are modeled as randomly choosing a subset of edges for a graph, and we can set the fraction of all possible edges to choose. For many graph properties, in particular for the 3-colorability in this thesis, it turns out that there is a special value called the *threshold* such that if the fraction we set is greater than the threshold, then asymptotically the probability of the property holding is one, while for a fraction less than the *threshold* the probability is zero. A similar phenomenon in physics is called a *phase transition*, a term used to describe any transition interval observed empirically when there is a sharp property change. Cheeseman, Kanefsky and Taylor [CKT91] have found that the average degree threshold for the graph coloring problem is 4.6.

This thesis will study a sequence of graphs generated throughout the *Frozen Development Process* (FDP) and these graphs are used as input to drawing algorithms. The FDP is defined as follows: given the vertex number N of a graph and a random permutation S of all $\binom{N}{2}$ vertex pairs, a series of graphs are constructed by adding the edges between the vertex pair in sequence. We denote G_i as the graph that has the first i edges for $i = 1, 2, ..., \binom{N}{2}$. Assuming that G_i is k-colorable, we check forward and call the vertex pair (u, v) *frozen same* if and only if $c(u) = c(v)$ for every valid coloring c of G_i. For a given sequence, we can determine a value m^* such that G_{m^*-1} is k-colorable but G_{m^*} is non k-colorable. We denote the *threshold* as the average over the set of all input sequences Π of the values m^*.

$$T(n) = \frac{1}{\binom{N}{2}!} \sum_{\pi \in \Pi} m^*(\pi).$$

Culberson and Gent [CG2000] found that, just before the threshold, there is a single edge which, when added, causes an average 16% of all vertex pairs to be frozen same. We call this phenomenon *catastrophic vertex collapse.*

We are interested in drawing the graphs throughout the FDP and animating the phenomenon of vertex collapse. Given an FDP, the input graph for the drawing algorithm is *collapsed graph at index* i, which is constructed from G_i as follows: we scan forward the remaining $(\binom{N}{2} - i)$ vertex pairs in S, and merge the pairs of vertices that have been frozen same by one of the first i edges.

1.3 Graph Editors and Layout Libraries

In this section, we will present some graph editors and layout libraries as well as our application.

1.3.1 Graphlet

Graphlet (see Figure 1.4) [MHB99] is a portable object-oriented toolkit for implementing graph editors and graph drawing algorithms. Graphlet is based on LEDA [MN99], a well-known library of the data types and algorithms of combinatorial computing.

Graphlet consists of the following components:

- The *core system* is the platform-independent part of Graphlet, and it contains the base and universal data structure and the algorithm interfaces.

- The *editor*. This is the main component of Graphlet, which is implemented in *LedaScript*.

- *Algorithm modules.* Since Graphlet is designed in a modular fashion, algorithms may be separate programs which can be written in C++ using LEDA.

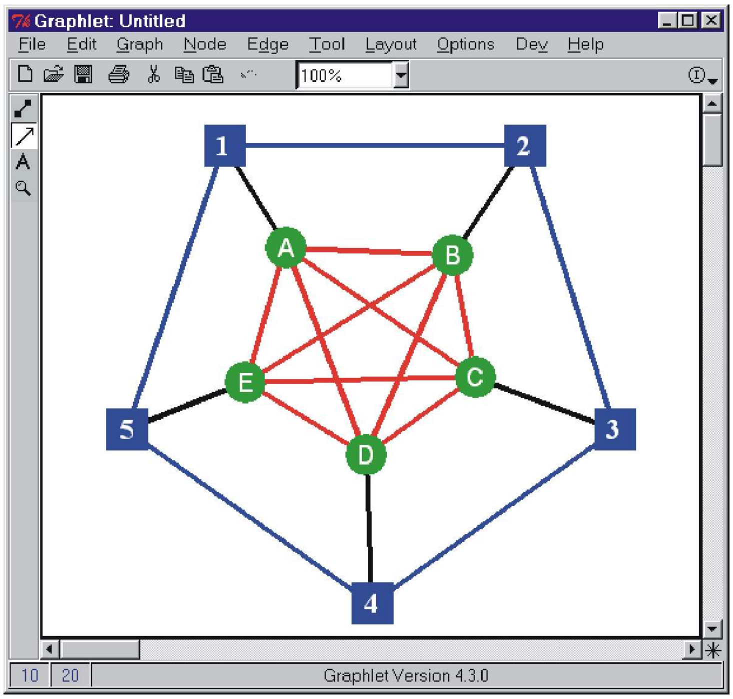


Figure 1.4: Demonstration of Graphlet [MHB99].

Graphlet also provides a portable graph language called GML (Graph Modeling Language). This is a file format for graph representation. External applications can communicate with Graphlet using GML files.

1.3.2 Graph Layout Toolkit

The Graph Layout Toolkit (GLT) [Tom2000] of the Tom Sawyer Software is composed of a graph management system, a portable drawing model and a virtual function layout system. A graph drawing produced by the GLT is demonstrated in Figure 1.5. The GLT provides the following four options:

- Circular layout (see Section 2.2). This layout emphasizes natural group structures and draws them in a circle model.
- Hierarchical layout (see Section 2.4). This layout assigns each node to a level to construct a hierarchy for the graph and generates a hierarchical drawing.
- Orthogonal Layout. This layout produces graph drawings in which edges

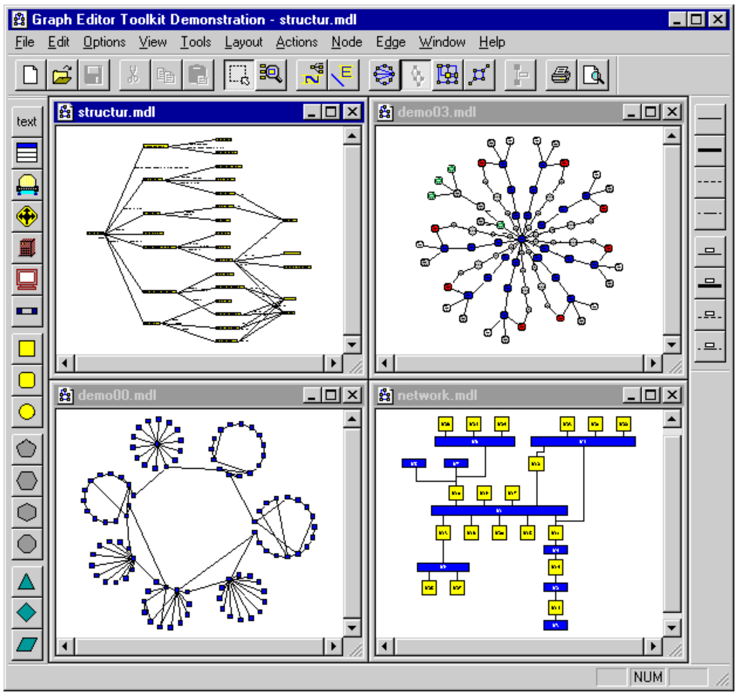


Figure 1.5: Demonstration of GLT [Tom2000].

are drawn parallel to the axes.

- Symmetric layout. This layout emphasizes the drawing aesthetics criterion of maximizing the symmetries of the graph.

1.3.3 Our Application: Graph Drawing with Clustering techniques (GDC)

GDC is a Java applet for drawing collapsed graphs and animating the vertex collapse phenomenon. The applet can be either run over the Internet or downloaded to run on a local site.

GDC has the following features:

- GDC can layout collapsed graphs of the FDP (see Chapter 4).

- GDC can animate the vertex collapse phenomenon of the FDP (see Section 5.2).

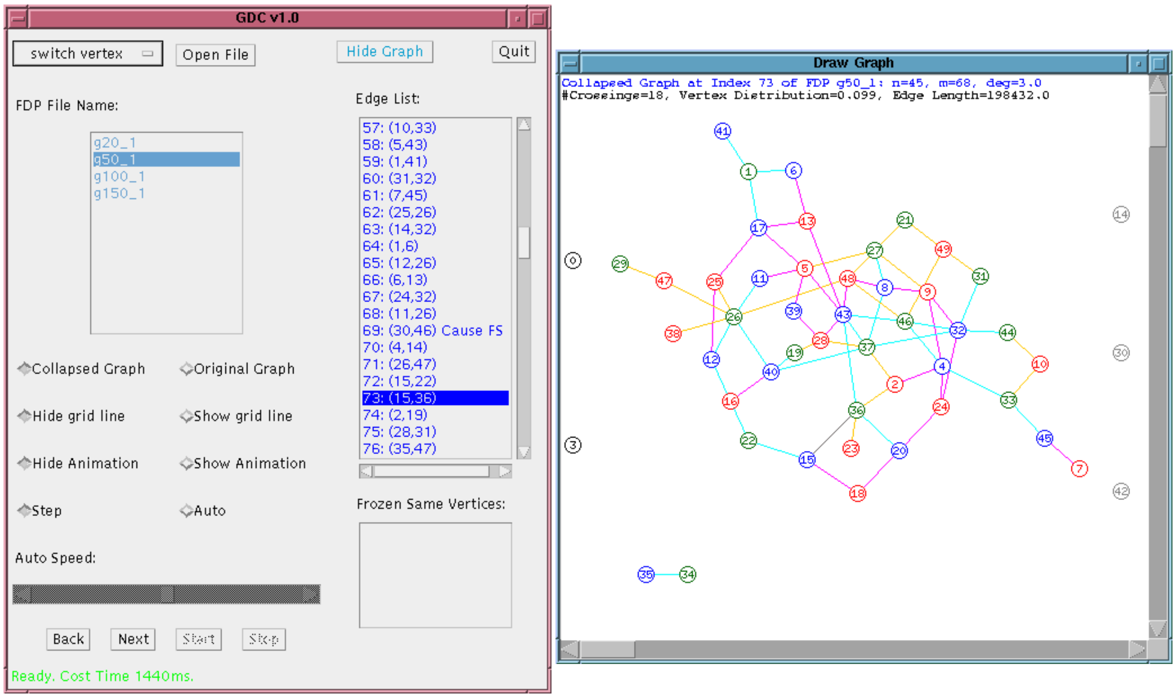


Figure 1.6: Demonstration of GDC.

- GDC provides drawing algorithms of 3 graph layouts: the circular layout (see Section 2.2), the spring layout (see Section 2.3) and the hierarchy layout (see Section 2.4). These algorithms are easy to reuse (see Section 5.1).
- It implements two clustering aided drawing algorithms (see Section 4.5).
- GDC can export its graph drawing to a *GML* format file, which can be further processed by other graph editors and layout libraries, such as Graphlet and VGJ [Stu96].

Chapter 2

Graph Layout

2.1 Introduction

Graph drawing algorithms are concerned with automatically generating visualizations for information spaces. The process of creating a graph drawing can be viewed as a three-stage pipeline (see Figure 2.1) [Kam89]. The first stage, modeling, extracts a graph from the information space. The graph is defined as $G = (V, E)$, where the vertices V $(= \{v_1, v_2, ..., v_n\})$ represent the abstract entities, and the edges E $(= \{e_{ij} = (v_i, v_j) | v_i \in V, v_j \in V\})$ represent the relationships of the entities in the information space. A graph is called *simple* if it contains no self-loop e_{ii} and no multi-edges e_{ij}. If $e_{ij} \neq e_{ji}$, the edge and the graph containing it are called *directed*; otherwise they are *undirected*. In this thesis, we will consider simple undirected graphs.

The *layout* stage of the pipeline will automatically assign a position for each vertex and a path for each edge in the visualization space. The input graph of this stage has no position, and it can have an infinite number of graph drawings. There is an intermediate phase called *graph embedding* in which we define the drawing order of edges around each vertex without considering the absolute positions of the vertices and paths of the edges. Obviously, a graph

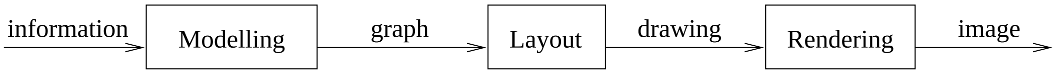


Figure 2.1: The Relational Visualization Pipeline.

Tree Layout	Reference	Comment
RT Tree Drawing	[RT81]	Places nodes on hierarchy levels and draws isomorphic subtrees with same appearance.
H-Tree Drawing	[Ead92]	All children of a node are placed on a common line, either horizontal or vertical.
Radial Tree Drawing	[Ead92]	Places nodes on concentric circles ; the common line is either horizontal or vertical.

Table 2.1: Drawing Methods for Tree Layout

Planar Layout	Reference	Comment
Testing planarity	[HT74]	Linear time complexity.
Planar Graph Drawing	[MM96]	Draws a planar graph on the basis of the algorithm in [HT74] .

Table 2.2: Drawing Methods for Planar Layout

embedding may be mapped to many graph drawings, but a graph drawing can be mapped only to one graph embedding. If each vertex of a graph drawing is positioned at the point whose coordinates are integers, this is called a *grid* drawing. In terms of the drawing style of edges, graph drawing algorithms can be divided into three classes: *straight-line*, *polyline*, and *curve*. Usually, the visualization space is either two-dimensional (2D) or three-dimensional (3D). In this chapter, we will mainly survey the straight-line drawing algorithms working in a two-dimensional rectangular plane, although these algorithms could be updated to work in a polygon or a three dimensional sphere by applying more heuristics.

Finally, the *rendering* stage produces an image of the graph drawing on a computer screen or on paper.

This chapter concentrates on the layout stage of the pipeline. The following five tables (Table 1,2,3,4,5) summarize some work on five graph layouts: tree layout, planar layout, circular layout, spring layout, and hierarchy layout.

There is much research on drawing methods especially designed for tree and other planar graphs, but they will be not discussed. We will describe a

Circular Layout	Reference	Comment
Makinen Algorithm	[Mak88]	Minimizes the circular dilation
GLT Circular Library	[DMM96] [RS97] [ESB99]	Clusters by biconnectivity, ratio cut or vertex distance and then positions clusters and the nodes in each cluster.
CIRCULAR Algorithm	[ST99]	Places the nodes along the longest path around a circle.

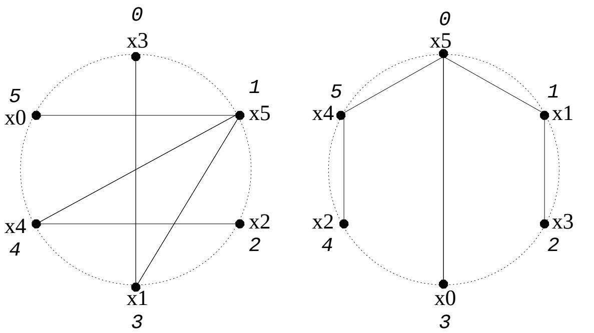
Table 2.3: Drawing Methods for Circular Layout

Spring Layout	Reference	Comment
Eades Spring Model	[Ead84]	Uses Hooke's law describing the forces between the nodes.
KK Spring	[KK89]	Force simulates the graph theoretic distance
Simulated Annealing (SA)	[DH91] [DH96]	Uses an energy model to optimize several aesthetic criteria simultaneously.
FR Force-directed	[FR91]	A variant of [Ead84] with electrical repulsive forces between vertex pairs.
GEM Force-directed	[FLM94]	A variant of [DH91] with attractive force towards the barycenter of a cluster.

Table 2.4: Drawing Methods for Spring Layout

Hierarchy Layout	Reference	Comment
Sugiyama (Hierarchy) Method	[STT81]	Orders the nodes on two consecutive layers to minimize the edge crossings using the barycenter of the neighbors.

Table 2.5: Drawing Methods for Hierarchy Layout



(a) Vertx Ordering: {5,3,2,0,4,1} (b) Vertx Ordering: {3,1,4,2,5,0}

Figure 2.2: Circular Layout Drawings of A Graph.

set of classic methods concerning three layouts: circular layout, spring layout and hierarchy layout.

2.2 Circular Layout

Circular layout methods place graph vertices along the circumference of a circle, and the edges are drawn as straight lines. The key point of circular algorithms is to find an *ordering* $f : V \rightarrow \{0, 1, ..., n-1\}$ for the vertices optimizing the number of edge crossings. We suppose that the vertex on the topmost of the drawing is ordered 0, and all the other vertices are ordered clockwise. Figure 2.2 presents two circular layout drawings for a common graph, and the ordering of vertices {x0,x1,x2,x3,x4,x5} is different for the two drawings.

The rest of this section is devoted to surveying the various circular drawing algorithms.

2.2.1 The Makinen Algorithm

Makinen [Mak88] proposed an algorithm attempting to minimize the number of edge crossings by minimizing the following formula

$$\sum_{(u,v)\in E} dilation(u, v) \qquad 2.1$$

where the $dilation(u, v)$ is defined as

$$dilation(u, v) = min(|f(u) - f(v)|,\ n - |f(u) - f(v)|).$$

For the drawing shown as Figure 2.2(a), Formula 2.1 is 12 and number of edge crossings is 4. Formula 2.1 of the drawing in Figure 2.2(b) is reduced to 7, and the number of edge crossings is reduced as well to zero. It has been proved that it is NP-Complete to minimize Formula 2.1 [MKNF87].

The Makinen algorithm includes two steps. First, two vertices with the highest degrees are placed at positions 0 and $n-1$, which correspond to the first position of the right and left halves in the drawing. Then the other vertices are processed as follows. We maintain the *left* and *right connectivity* array for all the vertices not yet placed, where the *connectivity* is the number of adjacent vertices already placed on the left (or right) half in the drawing. The vertex with the highest ($right\,connectivity - left\,connectivity$) will be placed on the right half. Similarly, the vertex with the lowest ($right\,connectivity - left\,connectivity$) will be placed on the left half. Ties are broken arbitrarily.

The drawing in Figure 2.2(b) can be produced by this algorithm.

2.2.2 The CIRCULAR Algorithm

The CIRCULAR algorithm [ST99] tries to reduce the edge crossings of a circular drawing by maximizing the number of edges appearing on the circular circumference. To achieve this, the CIRCULAR algorithm first removes one edge from every triangle subgraph in the cluster, then places the nodes in the longest path of a DFS (Depth-First-Search) tree along the embedding circle, and finally builds the corresponding vertex ordering.

2.3 Spring Layout

2.3.1 Introduction

Instead of being too concentrated on the fundamental theory of the nature of graph drawing, the drawing methods of spring layout are based on spring models that take advantage of other knowledge. They quantify the drawing objectives with optimization goals and constraints and further transfer the drawing problem to the CSP (Constraint Satisfaction Problem [Tsa93])

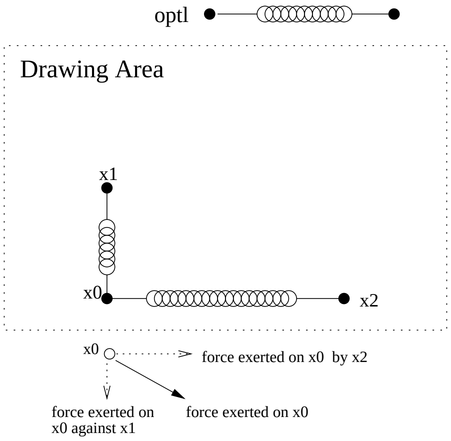


Figure 2.3: Spring Model.

searching problem.

The spring models (also known as force-directed models) were inspired by natural systems such as springs and macro-cosmic gravity. As shown in Figure 2.3, the drawing process simulates a mechanical system in which vertices are represented by particles, and edges are represented by springs (whose desired lengths are *optl* in this case). The particles are attracted by the springs if they are too far away and repelled if they are too close, and this is depicted as the force exerted on the particle x0 in Figure 2.3. The spring methods assume the straight-line style and formalize the searching objective as the equilibrium, where the sum of the forces on each particle is zero. There is a remarkable distinction between the force concerned here and the “force” in physics. The former force is used to calculate the velocity for the vertices during two static states of the drawing, but it does not involve the acceleration implied by the latter force.

The spring model may be defined as an energy system ([KK89][DH96]) rather than a force system ([Ead84][FR91]). In this case, the spring model may be viewed as assigning potential energy (based on springs and electrical energy) to a drawing. The algorithms are formalized to search for a drawing state in which the system energy is globally minimal.

There have been many spring layout methods developed since Eades first proposed the spring model in [Ead84], and every method is comprised of various heuristics. In general, heuristics have unpredictable output quality and time complexity, so the best way to evaluate them is do experiments with benchmarks, which are supposed to represent the typical problem instances [Eve99]. As a result of extensive experiments with several typical spring layout methods, Brandenburg, Himsolt and Rohrer [BHR95] concluded that there is no universal winner among these methods, and it is better to try several methods for a new application.

2.3.2 Schematic Form for Spring Layout Methods

Imitating the schematic form of the simulated annealing method ([DH96]), we construct a schematic form for spring layout methods within six steps.

1. Construct a spring model and quantify it with some aesthetic criteria.
2. Initialization: including parameters such as desired edge length and temperature for simulated annealing algorithms and initial node positions.
3. Select a node and move that node.
4. Cooling schedule for simulated annealing algorithms. Adaptive algorithms can adjust some parameters in this step as well.
5. Termination check. If the stop condition is not satisfied, go to stage 3.
6. Postprocessing: fine tuning.

In the rest of this section, we will discuss some interesting heuristics in each step, with the intention of finding ways to help reuse the heuristics from different steps to build new algorithms for special requirements.

Step 1. Construct a Spring Model and Quantify it with Some Aesthetics Criteria.

The spring models encode the desired drawing aesthetic criterion and their priorities into the formalized optimization objective for the methods to search for. In general, they can be divided into two types: the force model and the energy model.

(1.1) The Force Model

In the undirected graph drawing, various methods have modeled two basic types of force on an arbitrary node v:

- *spring force* f_{uv} towards its spring neighbor u. It is usually used to approach a drawing aesthetic criterion in which every spring has a desired edge length. The edges are generally modeled as springs [FR91], but some models suppose springs between non-adjacent vertex pairs [KK89].
- *repulsive force* g_{uv} against node u. This kind of force can prevent vertices from being placed too close to each other.

Since the computation of force law is usually the main cost of run time, extra care should be taken with the time efficiency of the force law. We will discuss some candidate force laws with respect to each force type.

(1.1.1) Spring Force Law.

Given a spring between vertex u and v, if the Euclidean distance $d(u, v)$ is equal to the desired distance $\delta(u, v)$, the spring is in a stable state, and there should be no spring force applied on v by u. This gives a basic case

$$f_{uv} \equiv 0, when\ d(u, v) = \delta(u, v).$$

An ideal spring force law is Hooke's classic law,

$$f_{uv} = k(d(u, v) - \delta(u, v))$$

where k is the spring stiffness. Although Hooke's law is time efficient, it does not treat extreme cases fairly, e.g. if u is either too far way from or too close to v. Eades [Ead84] suggested a logarithmic law

$$f_{uv} = k \ln(\frac{d(u, v)}{\delta(u, v)}).$$

The logarithmic law has the obvious disadvantage of a large computation effort. Fruchterman and Reingold [FR91] proposed a quadratic law

$$f_{uv} = k(\frac{d(u,v)^2}{\delta(u,v)} - \frac{\delta(u,v)^2}{d(u,v)})$$

which can achieve results similar to those of the logarithmic law. Based on experiments on several functions with powers of different orders, they found that the linear law $k(\frac{d(u,v)}{\delta(u,v)} - \frac{\delta(u,v)}{d(u,v)})$ tends to be easily trapped into local minima, and higher order functions do not work much better than the quadratic ones.

(1.1.2)Repulsive Force Law.

If there is no spring between vertex u and v, it will be possible that u and v are moved very close by simply applying spring force. This conflicts with the drawing aesthetic criterion of vertex distribution. Intuitively, the repulsive force against another vertex will be helpful. This brings up the problem of how to design a good repulsive force law.

As a complement of their force law, Fruchterman and Reingold [FR91] used the repulsive force law

$$g_{uv} = -k\frac{optl^2}{d(u,v)}$$

where $(u,v) \notin E$. The smaller the distance $d(u,v)$, the stronger the repulsive force exerted on v against u.

In the same paper, the authors pointed out that the repulsive force from distant vertices could be neglected in order to speed up the algorithm. An efficient way to determine whether u is *far* from v is as follows: geometrically divide the drawing plane into $n/4$ equivalent grids, assign each vertex a grid coordinate (e.g. $(row_v = 4,\ col_v = 3)$), and check whether $|row_u - row_v| \leq 1$ and $|col_u - col_v| \leq 1$.

There are some other types of forces for special graphs, such as

- *barycenter force* towards the barycenter of a group of vertices. When the group is comprised of the neighbors of the node, and if the input graph is triconnected and planar, the barycenter force can guarantee that the output drawing is convex [Tut63].

- magnetized spring force generated by a magnetic field through the drawing plane. The magnetic field can help restrain the orientation of the edges. For example, under a parallel magnetic field (where all magnetic forces have the same direction), the edges of a directed graph appear to be aligned along the field direction.

After the force on each node is computed, the candidate movement of a node is proportional to its force (see step 3). Another spring model is the energy model, which formalizes graph drawings with global energy functions, and step 3 will search a drawing with a local minimal energy.

(1.2)Energy Model

In general, energy models can be classified into two types: the force-based energy model and the general energy model. The former tends to find a stable state in which the Euclidean distance between vertices approaches a desired length. However, the force-based model doesn't consider some important drawing aesthetic criteria, such as minimizing the number of edge crossings. In the cases in which drawing quality rather than time efficiency is the main concern, the general energy model is a good choice because it considers many specified aesthetic criteria at the same time.

(1.2.1)Spring Energy Function

Ideally, the shortest path between u and v is spread along a line. Kamada and Kawai ([KK89]) defined the desired spring length $\delta(u, v)$as a function of *optl* and the shortest path distance $s(u, v)$, i.e.

$$\delta(u, v) = optl * s(u, v).$$

In their model, each vertex pair has a simulated spring between them. They modeled E_{uv}, the energy of v acted on by u, as the function

$$E_{uv} = \frac{1}{2}k(d(u, v) - \delta(u, v))^2$$

where, k, the stiffness parameter, is chosen by Kamada as

$$k = \frac{c}{\delta(u,v)^2}.$$

Here c is a constant. Thus, the energy function becomes

$$E_{uv} = \frac{c}{2}(\frac{d(u,v)}{\delta(u,v)} - 1)^2.$$

The global energy E is the sum of all the individual energies, that is

$$E = \frac{c}{2}\sum_{u,v\in V, u\neq v}(\frac{d(u,v)}{\delta(u,v)} - 1)^2. \qquad 2.2$$

(1.2.2) General Energy Function

The most popular general energy model was proposed by Davidson and Harel ([DH91][DH96]), and its global energy E is formalized as

$$E = \lambda_1 E_{vdistr} + \lambda_2 E_{border} + \lambda_3 E_{elength} + \lambda_4 E_{cross} + \lambda_5 E_{vedist} \qquad 2.3$$

where each term of E corresponds to the minimization of one drawing aesthetic criterion, and $\lambda_i\,(i = 1...5)$ is the priority for each aesthetic criterion:

- *Vertex distribution*: $E_{vdistr} = \sum_{u,v\in V}\frac{1}{d(u,v)^2}$. This term prevents vertices from coming too close together.
- *Borderlines*: $E_{border} = \sum_{v\in V}(\frac{1}{r_v^2} + \frac{1}{l_v^2} + \frac{1}{t_v^2} + \frac{1}{b_v^2})$, where r_v, l_v, t_v and b_v are the Euclidean distances from vertex v to the four borders (right, left, top, and bottom) of the drawing plane. It prevents vertices from being placed too close to any border of the drawing plane.
- *Edge Lengths*: $E_{elength} = \sum_{u,v\in V,(u,v)\in E} d(u,v)^2$. It prevents edges from being too long.
- *Edge Crossings*: E_{cross} is the number of edge crossings in the drawing. It aims at minimizing the edge crossings.

- *Node-edge distances*: $E_{vedist} = \sum_{v \in V, e \in E} \frac{1}{dist(v,e)^2}$, where $dist(v,e)$ is the perpendicular distance from vertex v to edge e. This term ensures that vertices are not positioned on the edges.

The General energy model is very flexible. More energy terms of drawing aesthetic criteria can be inserted into the model, and various assignments of priorities to the drawing aesthetic criteria can produce different solutions. Experimental results on the priorities assignment will be discussed in Chapter 3.

Step 2. Initialization

In this step, some parameters, such as *optl*—the desired Euclidean length of an edge, will be set. Also, the positions of vertices should be initialized in this step.

(2.1) Choose *optl*

Kamada and Kawai ([KK89]) determined that *optl* as the following formula:

$$optl = \frac{\sqrt{area}}{diameter}$$

where *diameter* is $max\{s(u,v)|u \in V, v \in V\}$—the largest length of the shortest path. This formula corresponds to the scenario in which the vertices on the longest one of the shortest paths are evenly placed along the drawing plane diagonal. If G is not connected, dummy edges can be added to make it connected. This avoids the extra computation of laying out all of G's connected components on the drawing plane.

Fruchterman and Reingold [FR91] calculate *optl* as

$$optl = c\frac{\sqrt{area}}{\sqrt{n}}$$

where c is a constant. Ideally, the vertices are distributed into exactly n different grid cells with the same size $\frac{area}{n}$. If the drawing plane is a square, the side of each grid cell will be $\frac{\sqrt{area}}{\sqrt{n}}$.

(2.2) Initial Vertex Positions

Most experiments confirm that the initial vertex positions have little influence on the output pictures, but they do affect the algorithm speed of converging to the equilibrium. Three normal initial placement ways are

- randomly place vertices;
- accept a manual initial placement.
- execute heavy-duty preprocessing. A simple example is to run another drawing algorithm first. D.Harel and M.Sardas [HS94] supplied a preprocessing algorithm particularly designed for a simulated annealing drawing method [DH91].

Step 3. Select a Node and Move that Node.

To find the equilibrium of a drawing, the algorithm should place each vertex at its local minimum position with respect to the force or energy function. When we select one vertex v to move and keep all the other vertices fixed at their locations, we quickly converge to a local minimum of v [BW97]. Iteratively repeating this step is a normal strategy to approach the global minimum [Ead84][KK89][FR91].

In the following section, we will explain how to determine which node to move and what direction that node be moved in and how far in order to converge to a local minimum rapidly.

(3.1) Select a Node to Move

There are three typical ways available

- randomly select a node to move [FLM94][DH91].
- sequentially select a node [FR91].

- select a node with the extreme value in terms of node metrics. Kamada and Kawai ([KK89]) introduced a node metric

$$\sqrt{(\frac{\partial E}{\partial x_v})^2 + (\frac{\partial E}{\partial y_v})^2}$$

where (x_v, y_v) is the two-dimensional coordinate of v's position; $\frac{\partial E}{\partial x_v}$ and $\frac{\partial E}{\partial y_v}$ can be computed by the formulas (7) to (8) as noted in [KK89].

(3.2) Determine a Movement for the Node

The movement selection plays an important role in spring drawing algorithms. In some ways, it is the kernel of all randomized layout algorithms. Based on the force and energy models (see step 1), various movement heuristics have been presented.

For the force model, the force on each vertex is

$$F(v) = \sum_{(u,v)\in E} f_{uv} + \sum_{(u,v)\notin E} g_{uv}$$

where f_{uv} is the spring force and g_{uv} is the repulsive force. Given v's position (x_v, y_v) (or (x_v, y_v, z_v) in $3D$ case), the x component of the force f_{uv} is $f_{uv} * \frac{x_v - x_u}{d(u,v)}$, and the x component of the force g_{uv} is $g_{uv} * \frac{x_v - x_u}{d(u,v)}$. The y (or z) component of f_{uv} and g_{uv} has a similar expression. Summing up the components of $F(v)$ along each axis, we can easily determine the movement vector for node v to neutralize F_v.

For the energy model, the global energy is either expressed by Formula 2.2 or 2.3. When a local minimum of 2.2 is found, the partial derivatives of E with respect to x_v and y_v should be zero, i.e.

$$\frac{\partial E}{\partial x_v} = \frac{\partial E}{\partial y_v} = 0.$$

The local energy minimum of v can be approached by iteratively moving vertex v using numerical methods (e.g. the *Newton-Raphson* method). Assuming that the position of v at the kth iteration is $(x_v^{(k)}, y_v^{(k)})$, $(x_v^{(k+1)}, y_v^{(k+1)})$ can be

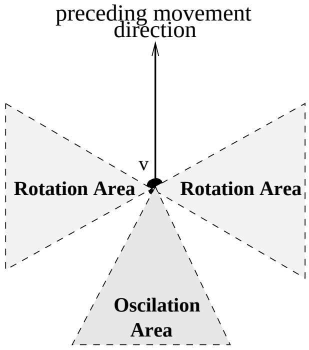


Figure 2.4: Direction of Rotation and Oscillation Areas [FLM94].

computed by solving linear equations whose coefficients are made of x_i^k and y_i^k for $i = 1, 2, ..., n$ (see the details at formulas (11) to (16) in [KK89]).

The optimization problem of energy Formula 2.3 is NP-hard, because even its individual term for minimizing the number of edge crossings is NP-hard [EMW86]. Assuming that after v is moved, the global energy is changed from $E1$ to $E2$, there is an efficient way to compute $E2$, e.g: count the updated aesthetic items due to v's movement, i.e. $E2 - E1$. A typical movement is to randomly select a vector length within a specified radius r (which usually becomes smaller as the temperature becomes lower). The angle of the movement vector should be chosen carefully. If the direction of movement is selected randomly, a vertex will rotate or oscillate around a position in some cases, because the random movement is unlikely to move that vertex out of a local minimum. Figure 2.4 [FLM94] demonstrates the rotation and oscillation area for the vertex v. Arne Frick et al. [FLM94] implemented a complicated rule to treat rotation and oscillation. Franz Brandenburg et.al [BHR95] suggested a simple rule called *re-enforcement*: if the preceding movement makes $E2 < E1$, the next movement is bound within almost the same direction, e.g. in the sector of width $\frac{\pi}{2}$.

General optimization strategies accept the movement if $E2 < E1$ but reject it if $E2 > E1$. However, they may be not suitable for minimizing Formula 2.3. Since the general model is intended to optimize several aesthetic criteria at the same time, v may have many local minima distributed among the drawing plane. To avoid v being entrapped into the local minima, a common optimization method used is Simulated Annealing (SA). The difference between SA and standard iterative improvement methods is that SA allows *uphill* movements—the next solution is worse than the current solution. SA can escape from local minima by applying the rule analogous to the physical process called *annealing*, in which liquids are cooled to a crystalline form. During the annealing process, the atoms will reach a thermal equilibrium at each temperature when the system energy obeys the Boltzmann distribution:

$$p(E) \simeq e^{-\frac{E}{kT}}$$

where $p(E)$ is the probability of the state of energy E, and T is the *temperature* and k is the Boltzmann constant [DH96]. When the system changes from the state of energy $E1$ to another state of energy $E2$ at temperature T, that probability is

$$e^{-\frac{E2-E1}{kT}}.$$

This formula suggests that when $E2 < E1$, the node v should be moved, and if $E2 > E1$, it is probabilistic. Kirkpatrick et al [KGV83] first modeled this annealing procedure for general optimization problems. The SA method has been applied successfully to spring drawing algorithms (e.g. [DH96][FLM94]).

During the movement process, a node may be moved against the drawing frame border. Usually a frame border is rectangular, but it can be a polygon, e.g. the shape of an island. This constraint can be solved by several heuristics [FR91].

Step 4. Cooling Schedule for Simulated Annealing Algorithms

For SA algorithms, the cooling schedule is used to adjust temperature T iteratively. The cooling schedule is a delicate part of the SA algorithm because it determines the ability of escaping local minima and the convergence speed. Many cooling schedules have been developed and studied in order to speed SA up. In general, the cooling schedule includes two parts. For the graph drawing problem, R.Davidson and D.Harel [DH96] proposed a cooling schedule as follows:

- Initial temperature. The algorithm sets the initial temperature high for the random initial vertex placement, but sets it low if the initial placement does not need much improvement.

- Temperature reduction. Like most researchers, they chose a generic rule. It is assumed T_k is the temperature at kth stage and $T_{k+1} = \gamma T_k$ where $0.6 \leq \gamma \leq 0.95$. At each temperature, $30 * n$ node movements are tried. Their experiments showed that more trials and a quicker temperature cooling were not helpful.

Step 5. Termination Check

Some methods set constant iteration numbers, e.g. the algorithm of [FR91] iterates 50 times, and the algorithm of [DH96] runs at 10 different temperatures. The other methods set stop conditions, e.g. the algorithm of [KK89] terminates when the values $\sqrt{(\frac{\partial E}{\partial x_v})^2 + (\frac{\partial E}{\partial y_v})^2}$ of all vertices are no more than a given boundary that is usually a small positive number.

Step 6. Postprocessing: Fine Tuning

The postprocessing phase can help refine the resulting graph. A typical fine-tuning method is to execute step 3 $c*n$ (c is a constant) times [FLM94][DH96].

Spring methods are widely implemented in the applications of various areas because they work for general graphs, and they are easy to understand and to program. They often produce highly symmetric drawings, distribute the vertices evenly, and keep the variance of the edge lengths small. Above all, they can be easily tuned by adding constraints to achieve the desired drawings, e.g., special forces can be introduced to restrict some vertices to be positioned within a given region.

2.4 Hierarchy Layout

We will introduce various hierarchy drawing heuristics with respect to different phases of the popular Sugiyama approach[STT81].

2.4.1 Introduction

Much research on straight-line hierarchy drawing has been done in the past two decades. Generally, the hierarchy layout methods transfer the input graph into a k-level *hierarchy* (also called *multilevel*) graph. If we denote V_i as the set of all the vertices on level i $(1 \leq i \leq k)$, we have $V = V_1 \cup V_2 \bigcup ... \bigcup V_k$. When a vertex v is in the level i (i.e. $v \in V_i$ or $rank(v) = i$), we define level $i-1$ as the *up-level* and level $i+1$ as the *down-level* of v or level i. We define the *ordering* of V_i as an assignment of indices $\{1, 2, ..., |V_i|\}$ to vertices in V_i.

The major aesthetic criterion of hierarchy layout methods is the minimization of the number of edge crossings. A normal approach of drawing a k-level hierarchy graph is to divide it into $k-1$ problems of minimizing the edge crossings of a 2-level graph. In this case, the ordering of one level vertices should be fixed, and the vertex ordering of the other level will be permuted in order to minimize the crossings. This problem is called the *one sided crossing minimization problem* or *Level Permutation Problem* (LPP). In other cases when both levels can change orderings, the problem is known as the *two sided crossing minimization problem* or *Bipartite Drawing Problem* (BDP). D.S.Johnson [GJ83] proved that BDP is NP-hard, and P.Eades and S.Whitesides [EW94] showed that LPP is NP-hard as well.

2.4.2 Sugiyama Approach

Among all the hierarchy layout methods, the most popular approach was designed by Sugiyama et al. in [STT81]. When applied to the straight-line drawing of general graphs, the Sugiyama approach consists of the following three phases:

- Phase 1: all the vertices V are partitioned into k different levels. In the hierarchy drawing, the y-coordinates of the nodes in a common level are usually same.

- Phase 2: the orders of vertices are permuted for each level to minimize the edge crossings while keeping the vertices on other levels fixed. This process is sequentially repeated for each level. This gives the relative x-coordinates of the vertices.
- Phase 3: the x-coordinates of vertices for each level are iteratively improved, while preserving the vertex ordering observed in phase 2.

Most hierarchy drawing heuristics are designed for a particular phase. In the rest of this section, we will follow this scheme to discuss some well-known heuristics with respect to each phase.

Phase 1. Partitioning vertices into layers.

For each vertex v, its level $rank(v)$ will be calculated by one of the following:

- using Breadth First Search (BFS).
- using Depth First Search (DFS).
- trying to minimize the *total edge length*; here the length of an edge is the number of layers it spans. The total edge length of G can be formulated as the formula:

  $$\sum_{(u,v)\in E} |rank(u) - rank(v)|.$$

  Then the problem is to find a ranking assignment to minimize this formula. One way is to solve an equivalent linear program in polynomial time [GKNV93]. Some complicated methods were proposed to find an optimal ranking for the edge weights [GKNV93].

When applying DFS or total edge length minimizing heuristics, we may add dummy vertices and edges to make sure every edge spans sequential levels. The updated graph is then processed in phases 2 and 3. According to Sugiyama's original scheme, phase 4 (which we do not discuss) will remove the dummy vertices and edges by generating long span edges. However, the edges will need to be drawn as polylines to preserve the number of edge crossings.

Although the ranking assignment produced by BFS cannot guarantee each edge spans nonsequential levels, it creates the problem that edges should be drawn between vertices on the same level. These edges can be drawn as arcs in a two dimensional drawing if the vertices from the same level are required to be drawn along a horizontal line.

Phase 2. Permute vertices on each layer to minimize the number of edge crossings.

(2.1) The DOWN-UP procedure

The drawing of a k-level graph $G = (V_1 \cup V_2 \bigcup ... \bigcup V_k, E)$ can be divided into $k-1$ LPPs of subgraph $G_i = (V_i \bigcup V_{i+1}, E)$. However, the LPP of G_i is related to the LPPs of G_{i+1} and G_{i-1}. The Sugiyama approach [STT81] suggested using the *DOWN-UP procedure*, which consists of a DOWN procedure and an UP procedure. The DOWN procedure processes $(k-1)$ two-level graphs in the order of $G_1, G_2, ..., G_{k-2}, G_{k-1}$, and changes the vertex positions (e.g. order or x-coordinate) of V_{i+1} on the basis of V_i for each G_i. Similarly, the UP procedure processes $(k-1)$ two-level graphs in the order of $G_{k-1}, G_{k-2}, ..., G_2, G_1$ and changes the vertex positions of V_i on the basis of V_{i+1} for each G_i. Each round of the DOWN-UP procedure involves $2 * (k-1)$ 2-layer subgraphs and a corresponding adjustment of vertex ordering of $2 * (k-1)$ levels, namely $V_2, V_3, ..., V_{k-1}, V_k, V_{k-1}, V_{k-2}, ..., V_2, V_1$. In practice, the algorithm can iterate the DOWN-UP procedures until it runs a given number of times or the properties of vertices appear periodically.

In the DOWN-UP procedure, each level is processed based only on either its up-level or down-level, whichever was processed in the preceding step. Sugiyama et al. tested another procedure in which each level is processed by considering both its up-level and down-level. In their examples, the DOWN-

UP procedure was always better[STT81].

Given a two-level graph $G = (V_1 \cup V_2, E)$ and a linear vertex ordering of V_1, the objective of LPP is to find a linear ordering of V_2 that minimizes the number of edge crossings. Since LPP is NP-hard, the algorithm efficiency of this phase determines the performance of the drawing method. In the past 20 years, therefore, many heuristics have been suggested for solving LPP.

Junger and Mutzel [JM97] empirically showed that the *barycenter* (BC) heuristic [STT81] and the *median* (ME) heuristic [EW86] significantly outperform some other LPP heuristics, e.g., split, greedy-insert, and greedy-switch heuristics [EK86].

(2.2) The Barycenter heuristic

For the hierarchy drawing problem, the barycenter heuristic was introduced in [STT81], where the barycenter specifies a vertex property derived from its neighbors on its adjacent levels. The barycenter property could be the vertex order, or the vertex x-coordinate. In this phase, the vertex's barycenter property is the vertex order, but the vertex x-coordinate is used in a similar barycenter heuristic of phase 3.

Given a two-level graph $G = (V_1 \cup V_2, E)$ and vertex ordering of V_1, the *index up-barycenter* ($upbc_{index}$) of any vertex $v \in V_2$ is defined as

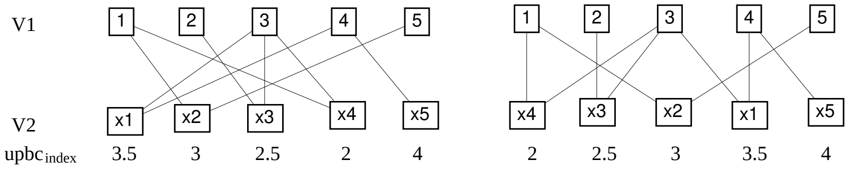
$$upbc_{index}(v) = \frac{\sum_{u \in V_1} a(u,v) * index(u)}{\sum_{u \in V_1} a(u,v)}$$

where

$$a(u,v) = \begin{cases} 1 & (u,v) \in E \\ 0 & (u,v) \notin E \end{cases} .$$

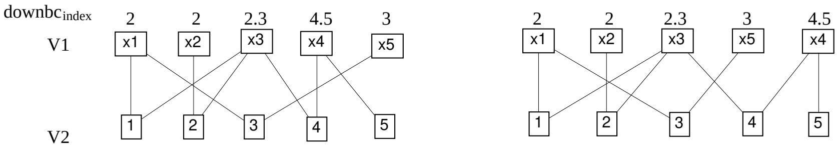
Figure 2.5(a) is the original graph, and (b) is the graph after ordering vertices in V_2 by their up-barycenter. This work reduces the number of edge crossings from 15 to 7.

Similarly, given a two-level graph $G = (V_1 \cup V_2, E)$ and a vertex ordering of V_2, the *index down-barycenter* ($downbc_{index}$) of any vertex $v \in V_1$ is defined



(a) before ordering V2 by up-barycenter (b) after ordering V2 by up-barycenter

Figure 2.5: A Demonstration of the up-barycenter.



(a) before ordering V1 by down_barycenter (b) after ordering V1 by down_barycenter

Figure 2.6: A Demonstration of the down-barycenter.

as

$$downbc_{index}(v) = \frac{\sum_{u \in V_2} a(u,v) * index(u)}{\sum_{u \in V_2} a(u,v)}.$$

Figure 2.6(a) is same as Figure 2.5(b), and Figure 2.6(b) is the graph after ordering vertices in level 1 by their down-barycenter. This work further reduces the number of edge crossings from 7 to 5.

For simplification, we define the bc_{index} as the $upbc_{index}$ in the DOWN procedure and the $downbc_{index}$ during the UP procedure. At each step of the DOWN-UP procedure, we compute the bc_{index} of the vertices in one level and then sort the bc_{index} in an ascending order and change the vertex's order with its index in the sorted bc_{index}. In case of a tied bc_{index}, it is a good practice to reverse the ordering[1] of tied nodes at the end of the current step [STT81].

Let us denote c_{vu} (resp c_{uv}) as the number of crossings of edges incident to u and v when we order v in front of u. Assuming $c_{vu} < c_{uv}$, we say that the barycenter ordering is *correct* if $bc_{index}(v) < bc_{index}(u)$.

The correctness of LPP heuristics can be evaluated by generating a matrix defined by [War77]. Sugiyama et al. [STT81] introduced a specific generating matrix to analyze the correctness of BC heuristics. The matrix is based on verifying the correctness of the barycenter ordering of many simple 2-layer graphs $G = (V_1 \cup V_2, E)$. Here, V_1 has r ordered nodes and its ordering is

[1]The tied nodes ordering is consistent with the preceeding ordering of this level.

known, and V_2 has only 2 nodes, $\{u, v\}$, which are randomly connected to V_1. The matrix emulates the $2^r * (2^r - 1)$ distinct graphs to check whether the barycenter ordering of $\{u, v\}$ is correct.

E.Makinen constructed an example in which $bc_{index}(v) < bc_{index}(u)$ but $c_{vu} > \lfloor\sqrt{r} - 2\rfloor * c_{uv}$ [Mak90]. On the other hand, the rate of correct ordering is remarkably high [STT81]: no more than 3.02% for r up to 10 [Mak90].

(2.3) The Median heuristic

Given a two-level graph $G = (V_1 \cup V_2, E)$ and a vertex ordering of V_1, for any vertex $v \in V_2$, we denote v's neighbor in V_1 as $(y_1, y_2, ..., y_t)$. Then the median heuristic (ME) defines

$$me(v) = rank(y_m)$$

where $m = \lceil \frac{t}{2} \rceil$. According to the ascending order of me, vertex ordering can be determined.

Although the correctness of ME is much less than that of the BC heuristic at least for r up to 10 [Mak90], Eades and Wormald [EW86] showed that $me(u) \leq me(v)$ implies $c_{uv} \leq 3c_{vu}$ for all pairs of vertices u and v. They further concluded that the number of edge crossings in the drawings produced by the ME heuristic is bounded and not more than three times that in the optimal drawing.

Phase 3: Positioning of vertices

Sugiyama et al. developed the Priority Layout method to improve the x-coordinates of vertices on each level according to an assigned vertex *priority* (e.g. vertex degree), while preserving the reduced number of edge crossings by satisfying the vertex ordering obtained in phase 2. Similar to the phase 2 method, this one improves the x-coordinates of vertices on each level with the DOWN-UP procedure and the barycenter heuristic. But instead of using vertex order, this phase chooses the horizontal position (i.e. x-coordinate) as the barycenter property. As with phase 2, there are two barycenters for each vertex with respect to the up-level and the down-level: the *position up-barycenter* ($upbc_{pos}$) and the *position down-barycenter* ($downbc_{pos}$). For a two-level graph $G = (V_1 \cup V_2, E)$ and any vertex $v \in V_2$, there is the following definition:

$$upbc_{pos}(v) = \frac{\sum_{u \in V_1} a(u, v) * pos(u)}{\sum_{u \in V_1} a(u, v)}$$

and for any vertex $v \in V_1$

$$downbc_{pos}(v) = \frac{\sum_{u \in V_2} a(u, v) * pos(u)}{\sum_{u \in V_2} a(u, v)}$$

where $a(u, v)$ was defined in phase 2. We define bc_{pos} as $upbc_{pos}$ in the DOWN procedure and as $downbc_{pos}$ during the UP procedure.

Phase 2 determines the vertex ordering by sorting the bc_{index}, and it has no other constraint. However, this phase aims to determine the x-coordinates (i.e. position) of vertices on each level by considering the following constraints in every step of the DOWN-UP procedure: (1) the vertex ordering should be preserved; (2) high-priority vertices select positions earlier than low-priority ones, and their positions cannot be changed prior to the end of this step; (3) a vertex should be positioned as close as possible to its bc_{pos}; (4) the position should be an integer between 1 and $width$, and no two vertices can take the same positions.

The Priority Layout method has three steps.

Step 1: Initialize the position for vertex v with the formula

$$pos(v) = offset + interval * index(v)$$

where $pos(v)$ is the x-coordinate of v, and $offset$ and $interval$ are integers.

Step 2: Iteratively execute step 3 to improve the positions of vertices in each level by following the DOWN-UP procedure. In each level, the vertex is processed in the sequence according to an assigned *priority*. Figure 2.7 demonstrates an example input graph for positioning the vertices. In this case, we chose degree as the vertex priority and set $offset = 0$ and $interval = 10$.

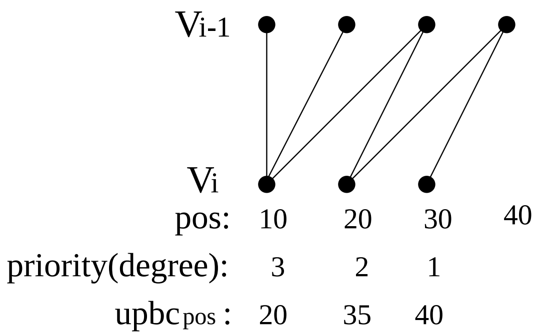


Figure 2.7: An Example of a Graph with Vertex *priority* and $upbc_{pos}$.

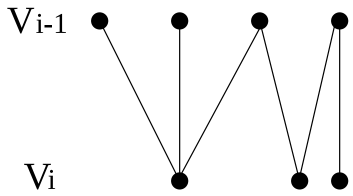


Figure 2.8: The Resultant Graph after Positioning V_i .

We will change the positions of vertex $v \in V_i$ in a DOWN procedure in the next step.

Step 3: At first, a new $pos(v)$ is selected. This $pos(v)$ should be close to $bc_{pos}(v)$ ($upbc_{pos}(v)$ in this case), and does not overlap with the positions of other vertices with higher priority. Then we may need to move some lower-priority vertices in order to preserve the vertex ordering derived from phase 2, but these movements should be as small as possible.

For example, the position tuple of vertices in V_i in Figure 2.7 will develop in the sequence as $(10, 20, 30)$, $(20, 21, 30)$, $(20, 35, 36)$, $(20, 35, 40)$, and then the graph becomes Figure 2.8.

2.5 Summary

In this chapter, we mainly investigate the graph layout. We have described some definitions related with graph layout, and presented some remarkable methods for the circular layout, the spring layout and the hierarchy layout. For each graph layout, we have attempted to characterize the drawing methods with a sequence of heuristics according to a framework. Many drawing algorithms introduced here have been realized in GDC, and the implementation details will be given in Chapter 3.

Chapter 3

The Collapsed Graph and the Drawing Algorithms' Implementation and Analysis

3.1 The Collapsed Graph

3.1.1 The Representation of Collapsed Graphs

Given an original graph G with N isolated vertices, we index these vertices with numbers from 0 to N-1. We then represent each edge (u, v) of the *Frozen Development Process* (FDP, defined in Section 1.2) of G with a four tuple $(v1, v2, edge\ type, frozen\ edge\ index)$, where $u \neq v$, $v1 = min(u, v)$, $v2 = max(u, v)$. The *frozen type* indicates whether $v1$ and $v2$ are *frozen same*[1] or not, and the *frozen edge index* points to the edge that causes $v1$ and $v2$ to be frozen same or records the current edge index. Table 3.1 lists the value domain for each field of the tuple.

We instantiate the k-Coloring problem with $k = 3$, which determines that

[1]It is defined in Section 1.2, and means that v1 and v2 have to be colored with the same color.

v1	v2	frozen type	frozen edge index
[0,N-2]	[1,N-1]	1: not frozen same 3: frozen same	[0, $\binom{N}{2}$-1] [0, T]

Table 3.1: Domain of the Fields in Edge Tuples of FDP

...	...	...	...
(8,	15,	1,	33)
(5,	17,	1,	34)
(**12,**	**14**,	3,	33)
(2,	18,	1,	36)
...	...	...	...

Table 3.2: A Sample Section of the Edge Tuples of an FDP.

all the vertices will finally fall into 3 clusters. In this sense, the process of vertex collapse in the FDP can be viewed as follows: initially each vertex is a cluster, and when two vertices are frozen same by the current edge, their corresponding clusters will merge. Thus, the vertices in a common cluster are kept frozen same with each other, and they should take the same color. In Table 3.1, T is the largest index of the edges that cause vertices frozen same. Our test graphs show $T < 5 * N$ for $N \leq 200$. In the CG_T (i.e., collapsed graph at index T, defined in Section 1.2), the vertex collapse terminates and a triangle is formed.

A sample section of the edge tuples of an FDP is given in Table 3.2. In this instance (8,15,1,33) indicates that the endpoints of edge 33—vertices 8 and 15—are not frozen same by any previous edge; (12,14,3,33) implies that vertices 12 and 14, the endpoints of this edge, have been frozen same by edge 33, i.e. the addition of edge (8,15) to the CG_{32} causes vertices 12 and 14 to be frozen same.

In order to compress the $\Theta(N^2)$ edge storage of the FDP, we represent the edges whose indices are greater than T by a hashtable in which the key (left column) is the edge index and the value (right column) is a subset of the clusters to be merged. Table 3.3 is a section of the hashtable of the FDP described above. To build up the hashtable, we first identify each cluster with the largest index of its vertices. We then apply the *union-find* algorithm [BG2000] to track the cluster merging process.

As we see in the table of edge tuples, vertex 12 is merged with vertex 14

...	...
30	**19, 14**
...	...
33	**19, 12**
...	...

Table 3.3: A Section of a Hashtable Describing the Information about Vertices to be Merged.

by edge 33. According to the hashtable above, vertex 14 has been merged to vertex 19 by edge 30, and the cluster is identified as 19. Since the cluster is identified as the largest index of the vertices inside, instead of "14, 12", "19, 12" is a value of key (edge) 33.

The number of rows in the hashtable is no more than $N-3$ because each row will reduce the number of clusters by at least one. Combined with the first T edge tuples, this hashtable can represent a collapsed graph with the focus on the vertex collapse infomation.

As the edge index i increases from 0 to $\binom{N}{2}$ throughout a complete FDP, each CG_i can be snapshot and laid out individually. There are three types of vertices in CG_i:

- *an unborn vertex*: a vertex that is not involved in the edges (i.e. vertex pairs) from 0 to i. Obviously, it is not merged and has degree 0.
- *a live vertex*: a vertex that occurs in the edges from 0 to i. The live vertex has not been merged to another vertex.
- *a dead vertex*: a vertex that occurs in the edges from 0 to i and has been merged to another vertex.

For example, CG_0 has one edge, N-2 unborn vertices, 2 live vertices and no dead vertices; CG_T has no unborn vertices, 3 live vertices and N-3 dead vertices. Both the unborn vertices and the dead vertices are isolated and easily visualized, so the induced subgraph of CG_i with the live vertices constitutes the input graph for the drawing algorithms. Thus, we represent CG_i with its

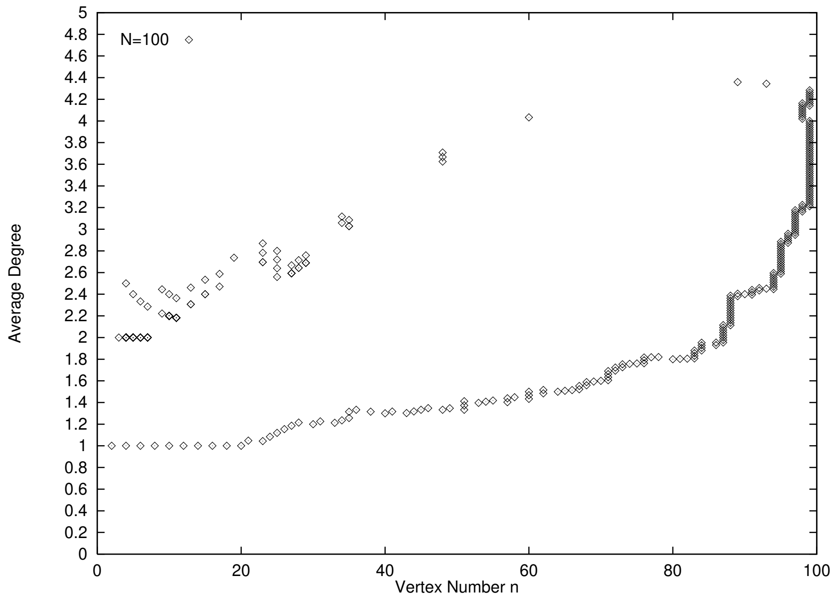


Figure 3.1: The Vertex Numbers and the Average Degree Domain of Collapsed Graphs for an FDP with $N = 100$.

live vertices and all of its edges. For each CG_i, we denote n as the number of the (live) vertices and m as the number of the edges .

3.1.2 Characteristics of Collapsed Graphs

In the FDP of an original graph G with N isolated vertices, the CG_i has different n's ($n \leq N$) and m's for each i.

Figure 3.1 shows the combinations between n and the average degree $\overline{deg}$ ($= 2 * m/n$) by plotting points. This figure clearly exhibits that the points form three parts:

- an almost continuous curve from the left-bottom to the right-top. The curve increases slowly when n is much less than N, and it becomes steep when n approaches N. These points correspond to the collapsed graphs before the *catastrophic vertex collapse* (defined in Section 1.2).

- several points sparsely distributed from the right-top to the middle-top. The catastrophic vertex collapse happens at this point.

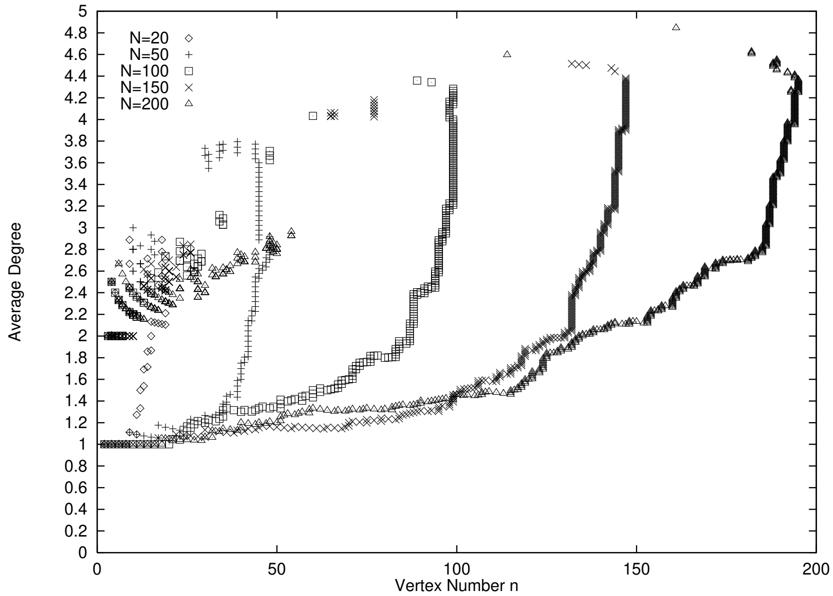


Figure 3.2: The Vertex Numbers and the Average Degree Domain of Collapsed Graphs. $N = 20, 50, 100, 150, 200$.

- a few points densely crowded in the left-middle area. These points depict the collapsed graphs as we aproach the final triangle (i.e. $n = 3$ and $\overline{deg} = 2$).

Figure 3.2 demonstrates 5 sample FDPs with different N's in one picture. This figure shows that the points of different FDPs have a similar distribution.

In order to evaluate the performance of the drawing algorithms, we construct the test set by choosing 13 sample collapsed graphs whose corresponding points in Figure 3.2 are listed in Table 3.4. These collapsed graphs are selected on the basis of N and $\overline{deg}$. A collapsed graph is called *small* if $N < 100$ and *large* if $N \geq 100$, and is called *sparse* if $\overline{deg} \leq 3.0$ and *dense* otherwise. We will use the tuple $(\overline{deg}, N)$ to identify each test graph in the thesis.

3.2 Introduction to the Evaluation of Drawing Algorithms

A good experimental comparison of graph drawing algorithms is the report [BHR95], which compared five spring layout algorithms (FR [FR91], KK [KK89],

N	20		50			100				150			
n	15	18	42	45	44	87	97	99	89	130	141	147	132
$\overline{deg}$	2.0	2.9	2.0	3.0	3.8	2.0	3.0	4.0	4.4	2.0	3.0	4.0	4.5

Graph Type	Test Graphs ($\overline{deg}, N$)
small and sparse	(2.0, 20), (2.0, 50), (2.9, 20), (3.0, 50)
small and dense	(3.8, 50)
large and sparse	(2.0, 100), (2.0, 150), (3.0, 100), (3.0, 150)
large and dense	(4.0, 100), (4.0, 150), (4.4, 100), (4.5, 150)

Table 3.4: N, n, $\overline{deg}$ and Type of Thirteen Test Graphs.

SA [DH91], GEM [FLM94], Tun [Tun94]) by evaluating two drawing metrics (the number of edge crossings, the ratio of the longest edge to the shortest edge) and the run time. Also, Davidson and Harel [DH91] and Fruchterman and Reingold [FR91] mutually compared their algorithms.

To compare our drawing algorithms and implementations, we mainly use the run time and two drawing aesthetic criteria—the number of edge crossings and the vertex distribution. They are defined as follows

- edge crossings $E_{cross} = \frac{1}{2}\sum_{e_1 \neq e_2 \in E} f(e_1, e_2)$ where $f(e_1, e_2) = 1$ when edge e_1 intersects with e_2 and $f(e_1, e_2) = 0$ otherwise. The aim is to minimize the number of edge crossings.

- vertex distribution $E_{vdistr} = \frac{1}{2}\sum_{u,v \in V, u \neq v} \frac{1}{d(u,v)^2}$ where $d(u, v)$ is the Euclidean distance between vertex u and v. This term aims to ensure that vertices do not come too close to each other.

We program the drawing algorithms using Java and try to minimize these aesthetic criteria and the run time for each algorithm. As an accessory drawing criterion, the minimization of edge length is introduced in order to balance the drawing aesthetics in our SA algorithm. The edge length is defined as follows

- edge length $E_{elength} = \sum_{(u,v) \in E} d(u, v)^2$. This prevents edges from being too long.

However, our observation of graph drawings suggests that minimizing the number of edge crossings and the vertex distribution aids human understanding more than minimizing the edge length.

3.3 The Implementation of Drawing Algorithms in GDC

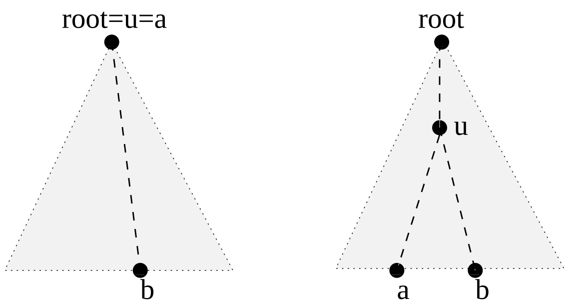
We implemented five different graph drawing algorithms: the Circular layout algorithm [ST99] (see Section 2.2.2), three spring layout algorithms–KK [KK89], FR [FR91] and SA [DH96] (see Section 2.3), and the Hierarchy layout algorithm [STT81] (see Section 2.4). In this section, we will describe some details of our algorithm's implementation.

3.3.1 The Circular Layout Algorithm

The circular layout is straightforward. Its special constraint is to force the vertices to be laid along a simple shape—the circumference of a circle. Our circular layout algorithm includes two steps: (1) it initializes the ordering of vertices using the CIRCULAR method [ST99] (see Section 2.2.2); (2) and it applies the greedy algorithm to swap vertices minimizing the number of edge crossings.

3.3.1.1 The Implementation of the CIRCULAR Algorithm

The CIRCULAR algorithm selects paths in the graph and places the nodes in the path along a circle's perimeter. This algorithm involves searching for the path with the longest length in a DFS (Depth-First-Search) tree. Here, the length of a path is defined as the number of vertices in the path. We denote $length(x, y)$ as the length of the path between vertex x and y. The longest path will either be from the root to a leaf or between two leaves (see Figure 3.3). Assuming that the longest path is from vertex a to b and vertex u is the vertex that is in the path and closest to the root, we can decompose the longest path into two paths: the one from u to a and another one from u to b.



(a) the longest path is between the root and a leaf
(b) the longest path is between two leaves

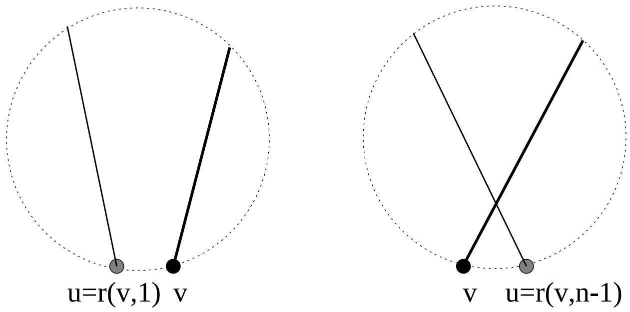
Figure 3.3: The Longest Path in a Tree.

To find u, a and b, we assign two three tuples (*length*, *leaf node*, *preceding node*) to each internal node v of the DFS tree in order to store the information about the top two longest paths from the *leaf node* to v. Here, the *length* is the path length from v to the *leaf node*, and the *preceding node* is the node such that it is in the path between the leaf node and v and its distance to the root is one more than that of v. The path information can be obtained by updating the information of the internal node based on that of its children, and this can be done in $\Theta(n)$ time. Thus, u is the vertex having the largest sum of *length* of the top two longest paths, and the two *leaf node*'s of u are a and b.

After the vertices in the longest path are placed along a circle, the other unplaced vertices are positioned as follows. We repeatedly find paths in the DFS tree and place the vertices in each path along the circle. Such paths can be found by scanning the DFS tree from an unplaced leaf node toward the root until it comes to a placed node.

3.3.1.2 The Greedy Algorithm that Iteratively Switches Vertices (GSV Algorithm)

Based on the initial ordering produced by the CIRCULAR method, we apply a greedy algorithm to swap vertices in order to reduce the number of edge crossings. We denote $r(x, l)$ as the vertex on the lth position from vertex x in the clockwise direction. In each trial, we randomly choose a vertex v and generate n different graphs, $G^{(0)}, G^{(1)}, ..., G^{(n-1)}$ where $G^{(i)}$ is constructed from



(a) Graph $G^{(i-1)}$ (b) Graph $G^{(i)}$

Figure 3.4: Switching Vertices Along a Circle.

$G^{(i-1)}$ for $i = 1, 2, ..., n-1$ by swapping v with $r(v, 1)$ (see Figure 3.4).

We compute the number of edge crossings in all of $G^{(0)}, G^{(1)}, ..., G^{(n-1)}$ and choose the graph with the minimum number as the initial graph for the next trial.

For graph $G^{(i)}$, we define $C^{(i)}$ as the number of edge crossings and $C^{(i)}_{uv}$ as the number of edge crossings in which u or v is involved.

We claim $C^{(i)} = C^{(i-1)} - C^{(i-1)}_{uv} + C^{(i)}_{uv}$.

Proof: We define $E_x = \{(x, y) \in E\}$ for $x \in V$. For both $G^{(i-1)}$ and $G^{(i)}$, we consider three subsets of E: E_u, E_v, and E_o=$E \setminus (E_u \cup E_v)$, and all edge crossings can be partitioned into four types: (1) an edge in E_u intersects another edge in E_v; (2) an edge in E_u intersects another edge in E_o; (3) an edge in E_v intersects another edge in E_o; (4) an edge in E_o intersects another edge in E_o. Only edge crossings of type 1 can be affected because just u and v change their relative orders. $\triangle$

When computing $C^{(i)}$ in a trial, we already know $C^{(i-1)}$. To compute $C^{(i-1)}_{uv}$, we use an auxiliary array A for v and define $A[k] = \sum_{j=2,3,...,k-1} P[v, r(v, j)]$ for $k = 3, 4, ..., n-1$ and $A[2] = 0$, where $P[x, y] = 1$ if $(x, y) \in E$ and $P[x, y] = 0$ if $(x, y) \notin E$. We have $C^{(i-1)}_{uv} = \sum_{k=2,3,...,n-1} A[k] * P[u, r(v, k)]$, which can be computed in $\Theta(n)$ time. Similarly, $C^{(i)}_{uv}$ can be obtained. This gives us a $\Theta(n)$ algorithm for computing $C^{(i)}$.

The GSV algorithm first computes the $C^{(0)}$ at the first trial by checking all edge pairs of the graph, and this costs time $\Theta(m^2)$. Then the GSV computes $C^{(i)}$ on the basis of $C^{(i-1)}$ $(i = 1, ..., n-1)$ for each trial. At the end of each trial, the GSV finds the graph with the minimum number of edge crossings in graphs $G^{(0)}, G^{(1)}, ..., G^{(n-1)}$ and uses it as the $G^{(0)}$ of the next trial. According to our experimental practice, the number of trials is chosen to be $\Theta(n)$. So the time complexity of the GSV is $\Theta(m^2+n^3)$, and that is $\Theta(n^3)$ for the collapsed graphs in which $m \in \Theta(n)$ (see Figure 3.2).

When the graph is large and sparse, we can further improve the time complexity of computing $C_{uv}^{(i)}$ using the efficient algorithm in Table 3.5.

Given graph $G^{(i)}$'s adjacency list and circular ordering of the vertices, arrays $X[0..deg(u)-1]$ and $Y[0..deg(v)-1]$ are generated such that $r(v, X[a]) \in N(u)$ and $r(v, Y[b]) \in N(v)$ for $a = 0, ..., deg(u)-1$ and $b = 0, ..., deg(v)-1$, where $N(x)$ is[2] the subset of vertices adjacent to vertex x and $deg(x)$ is the number of vertices in $N(x)$. We sort arrays X and Y in ascending order, and this time complexity is $\Theta(deg(u)*\lg deg(u)+deg(v)*\lg deg(v))$. We increase either a or b by 1 each time after comparing $X[a]$ with $Y[b]$, and this comparison complexity is $\Theta(deg(u)+deg(v))$. So this algorithm can compute $C_{uv}^{(i)}$ in time $\Theta(deg(u)*\lg deg(u)+deg(v)*\lg deg(v))$, compared with $\Theta(n)$ for the algorithm discussed earlier. For the collapsed graphs with $n \leq 200$, the $\overline{deg}$ is generally less than 5 (see Figure 3.2), so $\Theta(deg(u)*\lg deg(u)+deg(v)*\lg deg(v))$ tends to be a constant. In this case, the time complexity of the GSV algorithm is reduced to $\Theta(n^2)$.

Figure 3.5 is a graph drawing created by the GSV algorithm. The same collapsed graph ($N = 50$, $\overline{deg} = 3.0$) will be used to demonstrate the graph drawing of the other layout algorithms.

3.3.2 Spring Layout Algorithms

The spring layout algorithms have been widely implemented in many applications. In Section 2.3, we present some details of three typical spring layout

[2]If $(u, v) \in E$, this algorithm will remove v from $N(u)$ and remove u from $N(v)$.

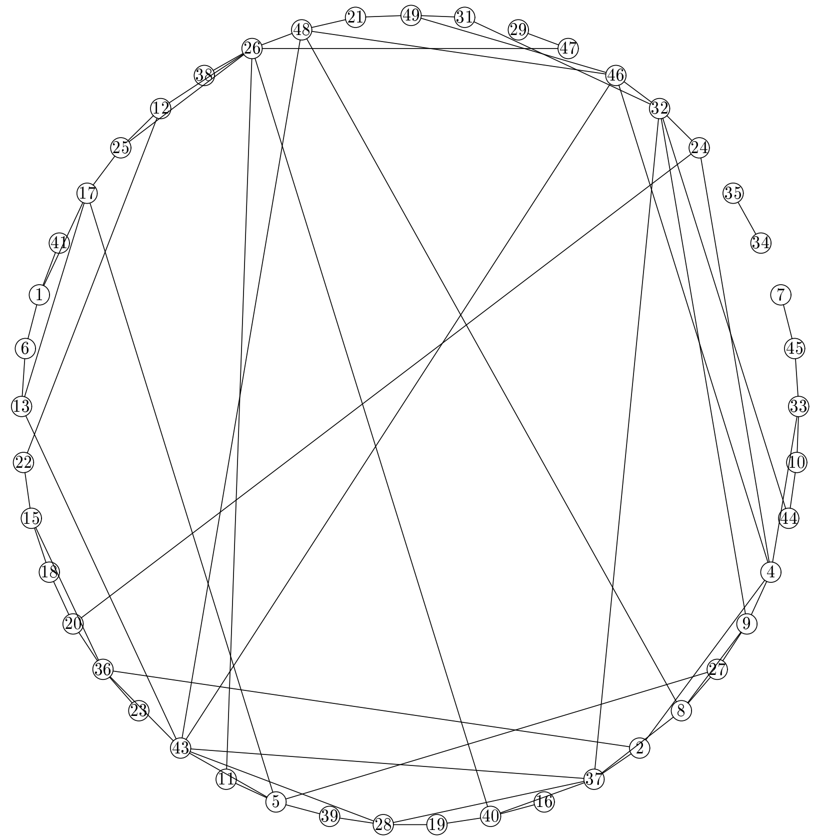
Input: Graph $G^{(i)}$'s adjacency list and circular ordering of vertices; u and v.
Output: $C^{(i)}_{uv}$.

Generate arrays $X[0..deg(u)-1]$ and $Y[0..deg(v)-1]$ whose entries store the distance from v to u's and v's neighbors.
Sort arrays X and Y in ascending order.
$a = 0$; a is the index of a u's neighbor
$b = 0$; b is the index of a v's neighbor
$p = 0$; p is the number of v's neighbors located between v and a u's neighbor
$C^{(i)}_{uv} = 0$;

```
do{
  if (b < deg(v)) /* at least one of v's neighbor has not been processed */
  {
    if (X[a] > Y[b]) /* r(v, Y[b]) is more close to v than r(v, X[a]) */
    {
      p = p + 1;
      b = b + 1;
    }
    else /* edge (u, r(v, X[a])) intersects p edges in which v is involved */
    {
      C_uv^(i) = C_uv^(i) + p;
      a = a + 1;
      if ( a > deg(u) - 1 ) /* all u's neighbors have been processed */
        break;
    }
  }
  else /* all v's neighbors have been processed */
  {
    C_uv^(i) = C_uv^(i) + p * (deg(u) - a);
    break;
  }
} while (true);

return C_uv^(i);
```

Table 3.5: An Efficient Algorithm for Computing $C^{(i)}_{uv}$.



Crossings=63, Vertex Distribution=0.098, Edge Length=1691927.0.

Figure 3.5: A Circular Layout Drawing of a Collapsed Graph Using *GDC.*

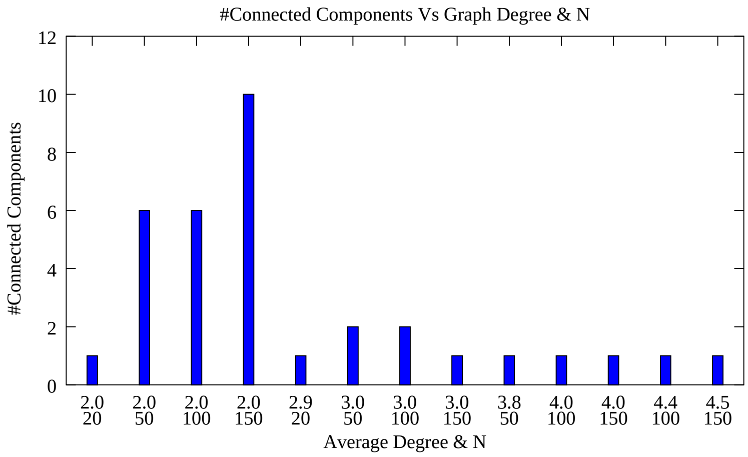


Figure 3.6: The Number of Connected Components Vs Graph $\overline{deg}$ and N.

methods: KK [KK89], FR [FR91] and SA [DH96]. Considering the properties of the collapsed graph and the applicability of the drawing heuristics, we implement our FR algorithm based on LEDA [MN99] and the KK algorithm based on VGJ [Stu96]. We build our SA algorithm by following the algorithm specification described in [DH96][FLM94]. In this section, we will introduce some interesting techniques applied in our implementation.

3.3.2.1 Making the Graph Connected

In all test graphs, five are not connected and the number of their components can be seen in Figure 3.6.

Figure 3.6 conveys that a graph with a low average degree is likely to be disconnected. This brings up the problem of how to lay out the connected components in one drawing plane so that no two components are entangled or too far away. An easy solution is to add auxiliary springs (dummy edges) between the connected components [KK89]. Assuming that the graph has k connected components $C_1, C_2, ..., C_k$ to be laid out, we add one edge between C_i and C_{i+1} for $i = 1, ..., k-1$ and one edge between C_k and C_1. Thus we make the graph connected. In the process of determining which vertex in each component is to be used as the endpoint of the dummy edge, we choose the one with the lowest degree in the component.

This preprocessing of the graph can be helpful in achieving better vertex

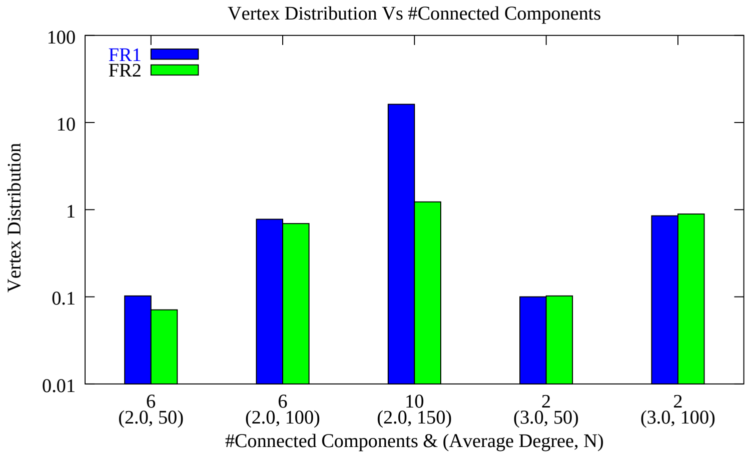


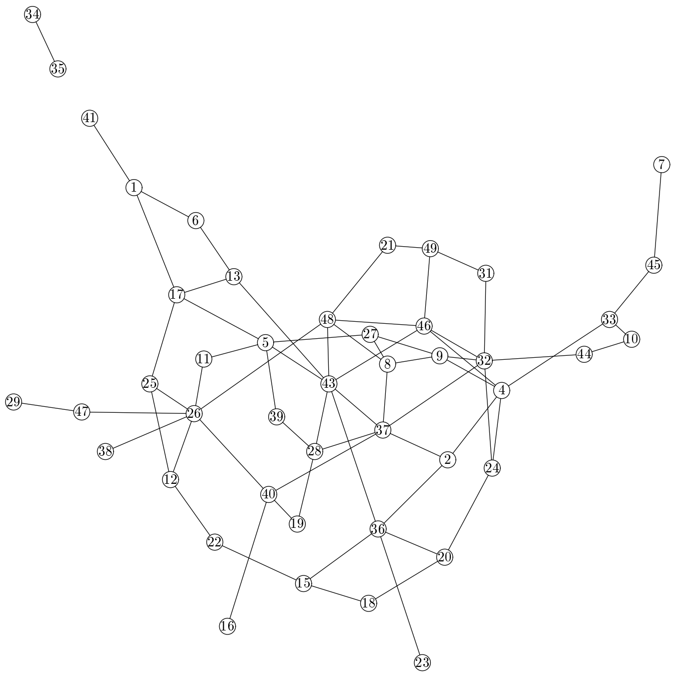
Figure 3.7: Vertex Distribution Vs the Number of Connected Components.

distribution with a trivial change in the run time and the number of edge crossings. In Figure 3.7, FR1 is the FR algorithm with no procedure to make the input graph connected and FR2 is the FR algorithm using that procedure. The figure shows that the greater the number of connected components, the smaller the vertex distribution that FR2 produces, as compared to that of FR1.

Figure 3.8, 3.9 and 3.10 present the graph drawings produced by spring layout algorithms for the same input graph in Figure 3.5. As we can see from these three figures, these force-directed algorithms assist in making every edge have same length and preventing vertices from being too close. The SA drawing is better than the other two, because it has smaller number of edge crossings and vertex distribution. Similar to the KK drawing in Figure 3.9, the FR one in Figure 3.8 has some vertices crowded closely, such as vertices 9,32,46 in Figure 3.8 and vertices 8,46,48 in Figure 3.9. According to the vertex distribution, the drawing in Figure 3.10 is a little better than those in Figure 3.8 and 3.9, and this may comply with most reader's perception.

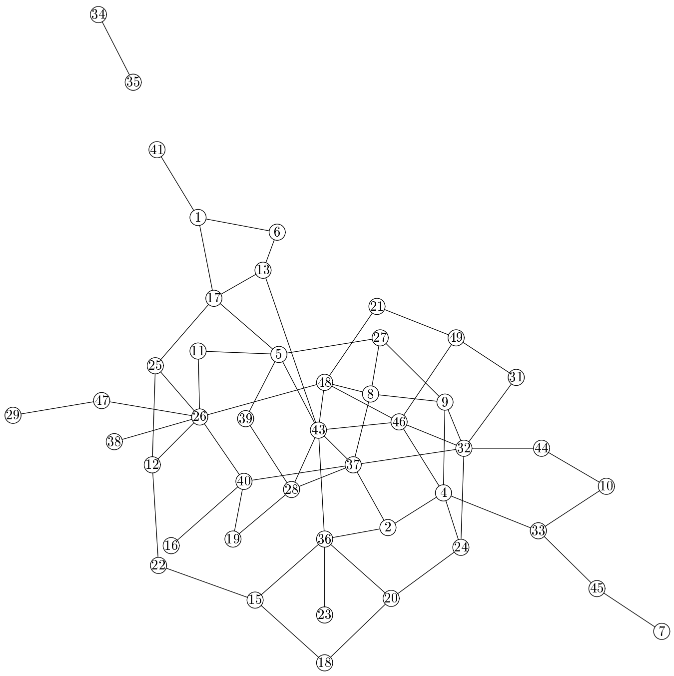
3.3.2.2 Efficient Computing of the Global Energy

In GDC, the global energy function (see Section 2.3) is $\lambda_1 * E_{cross} + \lambda_2 * E_{vdistr} + \lambda_3 * E_{elength}$ where λ_1, λ_2 and λ_3 are the weights of the drawing criteria. Each time after displacing a vertex, the SA algorithm needs to calculate the new



Crossings=24, Vertex Distribution=0.108, Edge Length=206822.0.

Figure 3.8: A Spring FR Layout Drawing of a Collapsed Graph Using *GDC*.



Crossings=19, Vertex Distribution=0.119, Edge Length=187297.0.

Figure 3.9: A Spring KK Layout Drawing of a Collapsed Graph Using *GDC*.

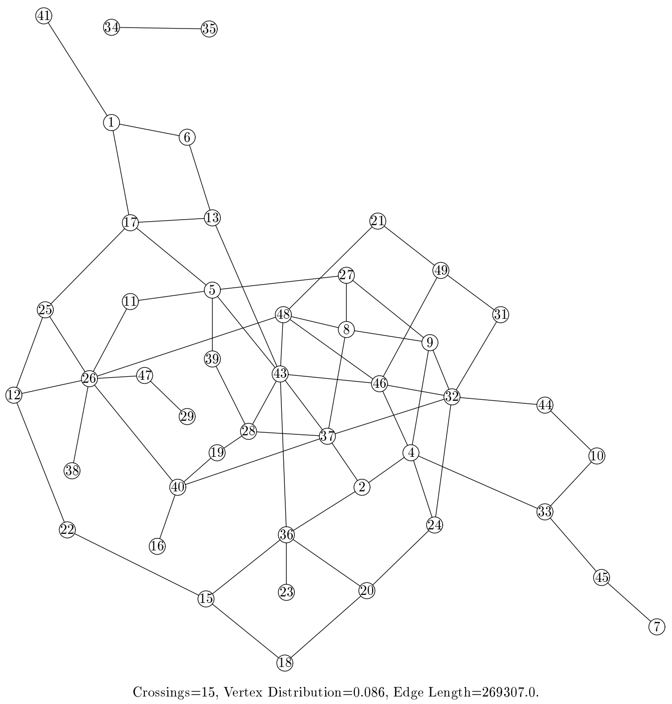


Figure 3.10: A Spring SA Layout Drawing of a Collapsed Graph Using *GDC.*

energy. Assuming that vertex v is selected to be moved, we call the vertex $v^{'}$ when v is moved to the new position. We define E_{cross}, E_{vdistr} and $E_{elength}$ as the values of the drawing criteria before v is moved and denote $E^{'}_{cross}$, $E^{'}_{vdistr}$ and $E^{'}_{elength}$ as the values of the criteria when v is moved to $v^{'}$. Given E_{cross}, E_{vdistr} and $E_{elength}$and the coordinates of v and $v^{'}$, we will show how to compute $E^{'}_{cross}$, $E^{'}_{vdistr}$ and $E^{'}_{elength}$ in an efficient way.

According to the definitions given above, we have

$$E^{'}_{vdistr} - E_{vdistr} = \frac{1}{2}\left(\sum_{u \in V, u \neq v'} \frac{1}{d(u,v')^2} - \sum_{u \in V, u \neq v} \frac{1}{d(u,v)^2}\right)$$

which can be solved in linear time. Similarly, there is $E^{'}_{elength} - E_{elength} = \sum_{u \in N(v)}(d(u,v^{'})^2 - d(u,v)^2)$. This is also cheap to calculate. So the computation efficiency of the energy function is dependant on the procedure of computing $E^{'}_{cross}$ directly or calculating $E^{'}_{cross} - E_{cross}$.

When computing $E^{'}_{cross}$ without using E_{cross}, we can use the *uniform grid* technique [Tun94] that works best in those cases that the force-directed algorithms tend to produce in which the vertices are distributed evenly in the

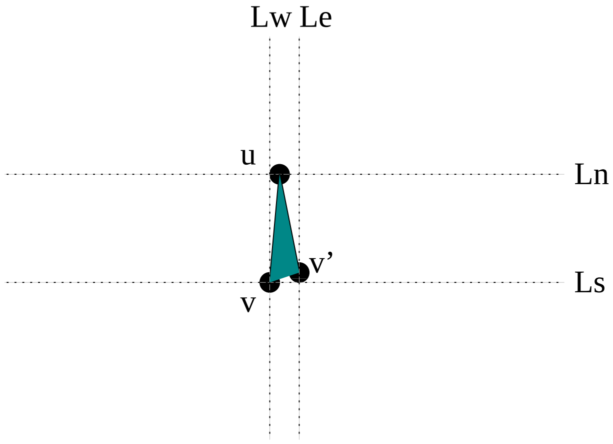


Figure 3.11: Computing the Number of Edge Crossings after Moving v to $v^{'}$.

drawing plane. Given a $g \times g$ uniform grid to partition the drawing plane, the best time complexity of the uniform grid algorithm is $O(n \log(g) + (n/g^2)^2)$ [ASB94], where g should be chosen to be not too large and not too small.

Instead of calculating $E^{'}_{cross}$ directly, we try another efficient way to compute $E^{'}_{cross} - E_{cross}$ in GDC. As shown in Figure 3.11, u is a neighbor of v, and u, v and $v^{'}$ form a triangle that is displayed in shadow. In most cases, the distance between v and $v^{'}$ is much less than the width or the height of the drawing plane, and the area of the shadow is very small compared with the complete drawing area, in particular when n becomes large. To compute $E^{'}_{cross} - E_{cross}$, we need only consider the edge crossings that involve either edge (u, v) or edge $(u, v^{'})$ for all $u \in N(v)$. Given edge (u, v) and an edge not adjacent to v, if both endpoints of this edge have been determined to be (1)west of Lw, or (2)east of Le, or (3) north of Ln, or (4)south of Ls (see Figure 3.11), this means that this edge does not intersect with edge (u, v) or $(u, v^{'})$. So we do not have to run the time-consuming procedure of checking the edge intersection, although the number of edges to be checked against edge (u, v) is still $m - 1$. Thus, we can save the effort of computing $E^{'}_{cross} - E_{cross}$, which is especially effective when the graph is large.

3.3.3 The Hierarchy Layout Algorithm

Most hierarchy layout methods apply the Sugiyama strategy [STT81] (see Section 2.4). Based on the pseudo code of the Sugiyama algorithm presented by Ivan Bowman [Bow98], we implement our hierarchy layout algorithm, which has two phases, as follows:

- Phase 1. Partition vertices into layers. We assign each vertex to the layer of the level on which it appears in the BFS (Breadth First Search) tree. This prevents the edge from crossing nonadjacent layers, but brings up the problem that edges can connect with vertices on the same layer.

- Phase 2. Permute vertices on each layer to minimize the number of edge crossings. We use the *barycenter heuristic* combined with the *DOWN-UP procedure* (see Section 2.4) to iteratively order the vertices on one level according to their *order down-* or *up-barycenter* (defined in Section 2.4). We reverse the order of nodes with equal barycenters to break the ties. For those edges appearing on one layer, we draw them with arcs instead of straight-lines and apply the greedy algorithm to swap nodes on the given layer to reduce the number of edge crossings in which both the edges within that layer and the edges between distinct layers are involved.

The hierarchy layout drawing of the collapsed graph demonstrated in Figure 3.5 (and 3.8, 3.9, 3.10) is displayed in Figure 3.12.

Figure 3.12 demonstrates the display window of the graph drawing in our application GDC. The top row of text on the window presents some information concerning the collapsed graph, e.g., the FDP name "50_1" implies this FDP has $N = 50$ and "73" indicates the current edge index. The second row of text gives three drawing criteria that are further used for evaluating the algorithm's performance. The graph drawing pane is divided into three parts: the left one is used for drawing unborn vertices, the right one for dead vertices, and the middle one for the live vertices and all the edges of the collapsed

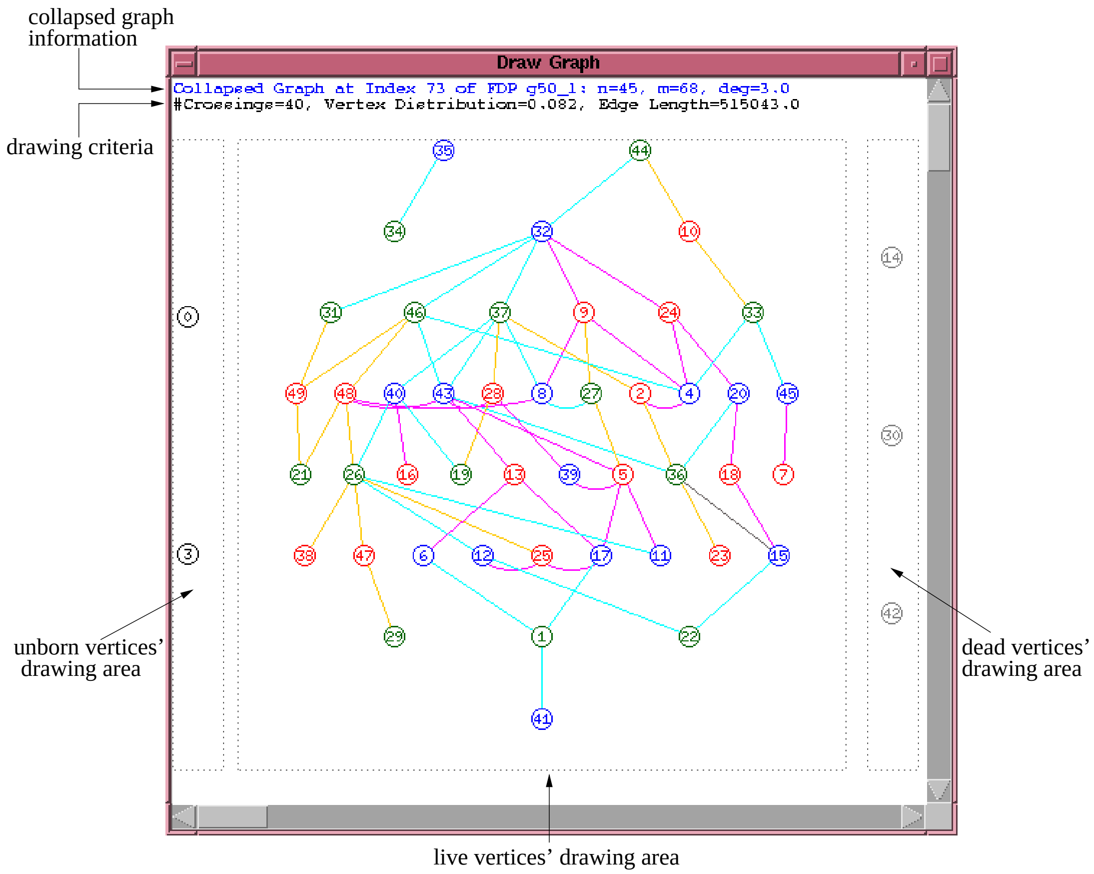


Figure 3.12: A Hierarchy Layout Drawing of a Collapsed Graph Using *GDC*.

graph.

3.4 Experimental Results and Analysis

In this section, we will first analyze the relationship between the quality of the drawing criteria and their weights in SA. Then we will evaluate the drawing algorithms (Circular, KK, FR, SA and Hierarchy) of various layouts (Circular, Spring and Hierarchy) in terms of the number of edge crossings, the vertex distribution and the run time .

All the experimental results (in both Chapter 3 and Chapter 4) are obtained based on the assumption that all the drawing algorithms have the same drawing plane, the same running environment (Dual Pentium III 450M CPU/256M Memory/Linux 2.2.10), a common test graph set (see Table 3.4), and 50 trials per test graph.

3.4.1 Experimental Results and Analysis of the SA Algorithm

In Section 3.3.2.2, we have introduced the SA parameters λ_1, λ_2 and λ_3, which represent the weights of three drawing criteria: edge crossings, vertex distribution and edge length. The minimization of edge crossings is an important objective of graph drawing, and the vertex distribution should be minimized at the same time. However, the cost function composed of these two criteria will not work at the low temperature during the annealing process, when the vertex distribution is likely to dominate such cost function and forbid new vertex movement. Thus the minimization of the edge length is introduced to balance the vertex distribution in order to get out of local minima. Unlike the cost function used in [DH96], ours do not contain another two drawing criteria: *borderlines* and *node-edge distance* (defined in Section 2.3.2). The minimization of borderlines is discarded because it is not useful when the vertices are not initially placed on the border of the drawing area. We get rid of the minimization of the node-edge distance because its effect has been covered

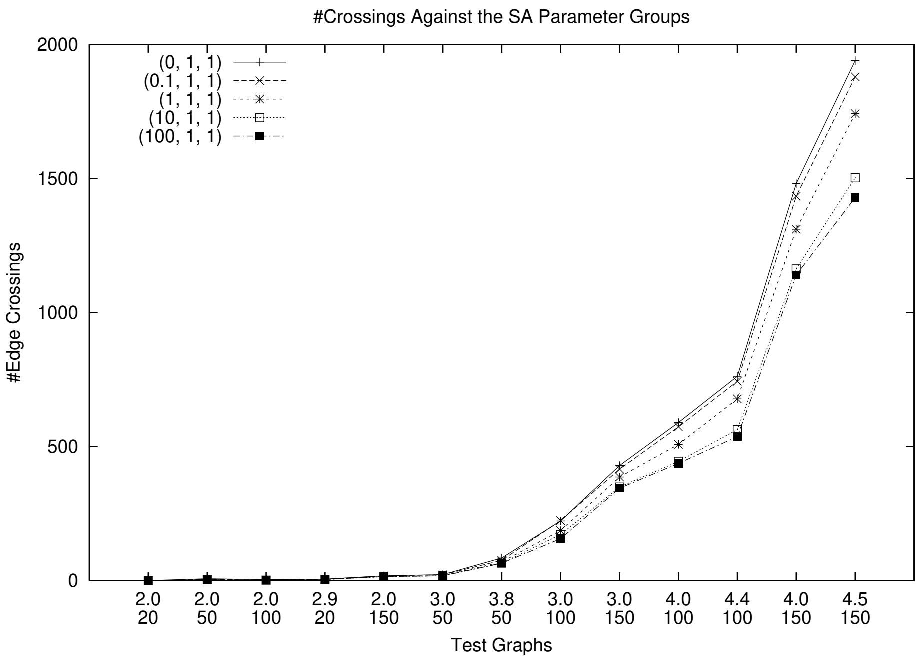


Figure 3.13: Number of Edge Crossings Against the SA Parameter Groups.

by the process of minimizing the vertex distribution and the edge length.

The configuration of these weights can affect the drawing criteria. First, we choose five groups of SA parameters $(\lambda_1, \lambda_2, \lambda_3)$ for evaluation, and they are (0,1,1), (0.1,1,1), (1,1,1), (10,1,1) and (100,1,1). These parameter groups stand for a sequence of increasing priorities of the edge crossings in the objective function with fixed λ_2/λ_3 (=1). In the experiments, the weight parameters λ_i are normalized and all the SA algorithms with different parameter groups use the same initial graph drawing. Figure 3.13 presents the edge crossings against these groups of SA parameters. Generally, the larger $\frac{\lambda_1}{\lambda_1+\lambda_2+\lambda_3}$ is, the smaller the number of edge crossings will be.

Secondly, we compare another five parameter groups: (1,1.1,1), (1,1,0.9), (1,1,1), (1,0.9,1) and (1,1,1.1). The λ_2/λ_3 is 1.1 for the former two groups, 1 for the third one and 0.9 for the latter two. These parameter groups are verified to produce reasonable graph drawings in practice[3]. The λ_2/λ_3 is bounded between 0.9 and 1.1 in our experiments, because the number of crossings tends

[3]As a counterexample, (0,0,1) produces a graph in which vertices are too crowded to be distinguishable.

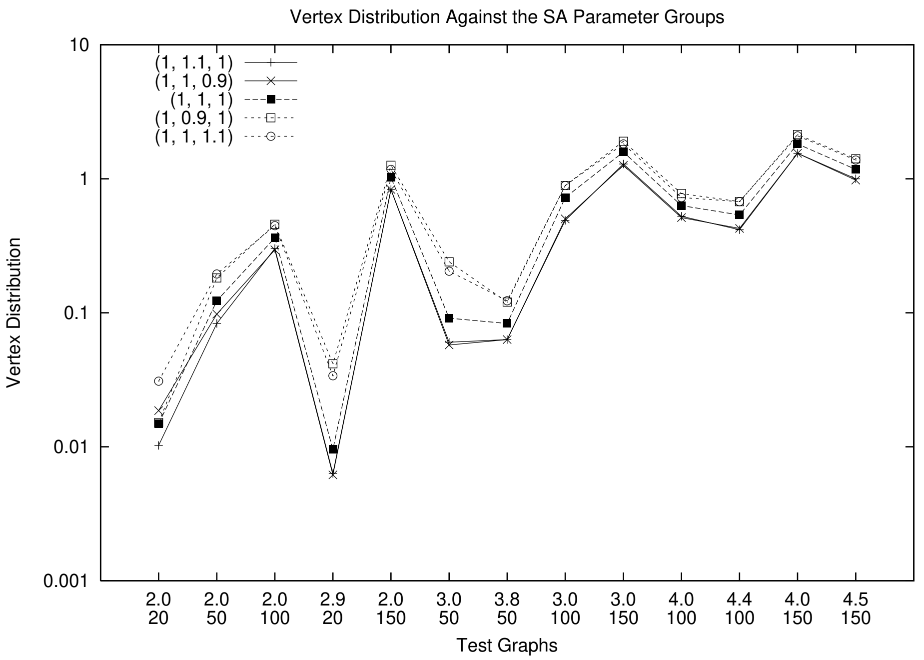


Figure 3.14: Vertex Distribution Against the SA Parameter Groups.

to be high when λ_2/λ_3 is large and the vertex distribution is likely to be ruined when λ_2/λ_3 is small.

Figure 3.14 and 3.15 present the vertex distribution and the edge length against the SA parameter groups respectively. Obviously, in both figures, the curve of (1,1.1,1) almost overlaps that of (1,1,0.9). Similarly, the curve of (1,0.9,1) almost overlaps that of (1,1,1.1). As regarding the vertex distribution, (1,1.1,1) and (1,1,0.9) work better than (1,1,1), which is better than (1,0.9,1) and (1,1,1.1). This complies with the increasing order of λ_2/λ_3. In the contrast, the ordering of parameter groups is reversed in terms of the edge length. According to Figure 3.15, (1,0.9,1) and (1,1,1.1) perform better than (1,1,1), which is better than (1,1.1,1) and (1,1,0.9). So we can say the larger λ_2/λ_3 is, the smaller the vertex distribution is and the larger the edge length is.

As described before, the calculation of crossings is time consuming. Compared with other parameter groups, (0,1,1) always requires much less run time because it avoids computing the number of edge crossings. This can be seen in Figure 3.16, which shows the run time of our SA algorithm with or without

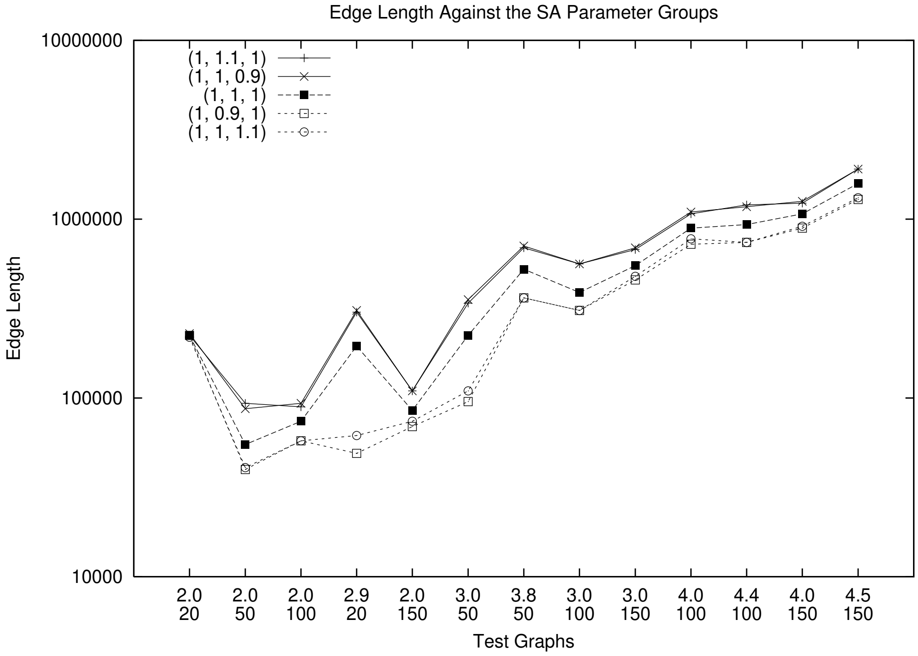


Figure 3.15: Edge Length Against the SA Parameter Groups.

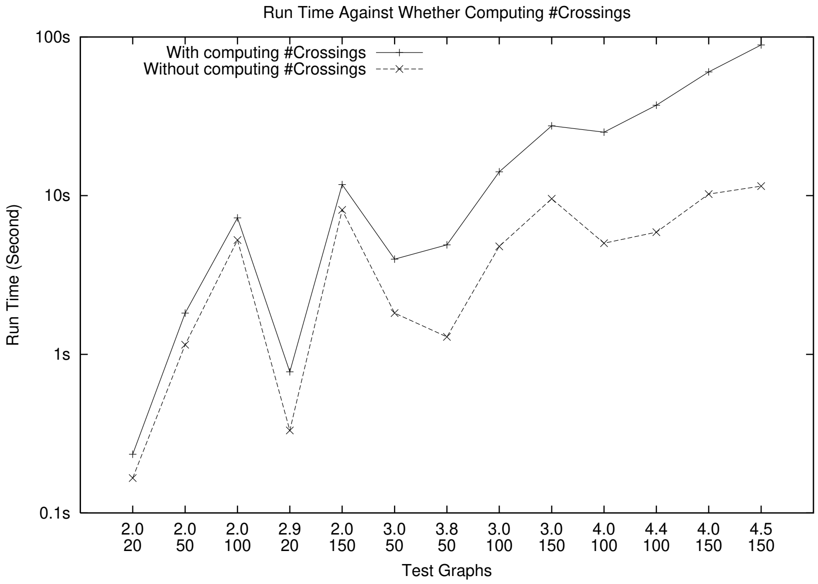


Figure 3.16: Run Time Against Whether Computing the Number of Crossings.

computing crossings for all test graphs.

Finally, we conclude with the relationship between λ_1/λ_3 and λ_2/λ_3 based on our experimental results. When $(\lambda_2/\lambda_3) \geq 1$, the larger $\frac{\lambda_2+\lambda_3}{\lambda_1+\lambda_2+\lambda_3}$(i.e. $\frac{\lambda_2/\lambda_3+1}{\lambda_1/\lambda_3+\lambda_2/\lambda_3+1}$) is, the larger the number of crossings is and the smaller the vertex distribution is. Since (1,1,1) can achieve balanced crossings and vertex distribution at the same time, it is chosen as the parameter group for our testing of SA algorithm.

3.4.2 Experimental Results and Analysis of Different Layout Algorithms

In this section, we will evaluate the drawing algorithms according to their performance, namely the number of edge crossings, the vertex distribution and the run time. The edge length will not be discussed because it appears to be not strongly related to the readability of the graph drawing according to our observations.

While testing the drawing algorithms, we start SA with a good initial drawing—the output of FR—and initialize the other drawing algorithms with the simple node placement in which all the vertices are laid along a circle in the order of their indices.

Figure 3.17 presents the number of crossings of the drawing algorithms. For those graphs whose $\overline{deg} = 2.0$, the numbers of edge crossings is zero or very small, implying that they are likely to be planar graphs. In contrast, the dense graphs are probably not planar.

For convenience in representing the ranking of the drawing algorithms, we use "algorithm1<algorithm2" to demonstrate that algorithm1 is much better than algorithm2 and "algorithm1≤algorithm2" to present that algorithm1 is a little bit better than algorithm2 in most cases with respect to the criterion in use. Thus, according to Figure 3.17, we can represent the ranking of algorithms with Table 3.6.

The table suggests that the spring layout algorithms are better than the

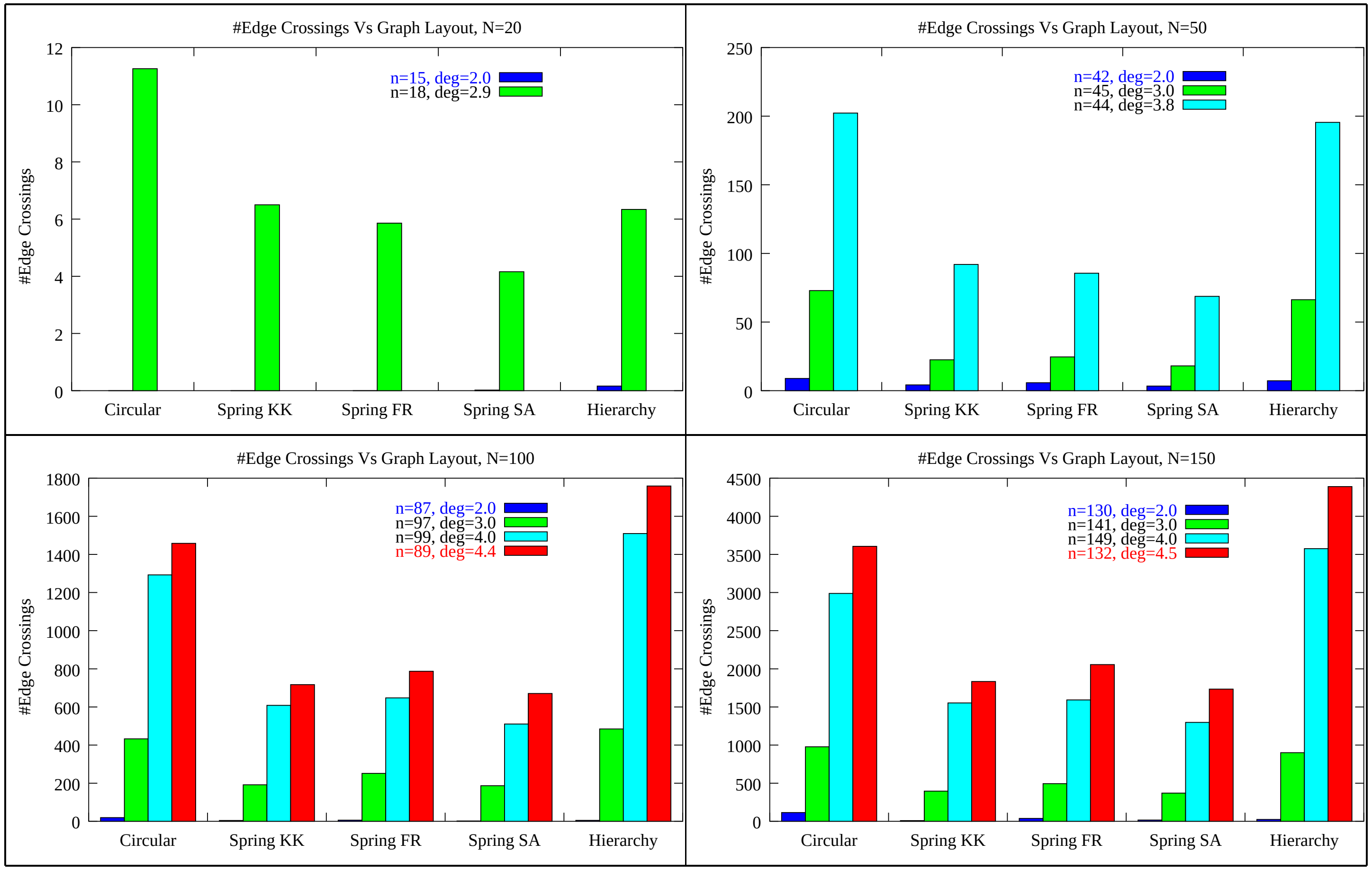


Figure 3.17: Number of Edge Crossings Versus Collapsed Graphs.

Collapsed Graphs	Algorithm Ranking w.r.t #Edge Crossings
small	SA $<$ FR $\leq$ KK $<$ Hierarchy $\leq$ Circular
large	SA $<$ KK $<$ FR $<$ Circular $<$ Hierarchy

Table 3.6: Algorithm Ranking With Respect to the Number of Edge Crossings.

Collapsed Graphs	Algorithm Ranking w.r.t Vertex Distribution
small	Circular $<$ Hierarchy $<$ SA $<$ KK $\leq$ FR
large and sparse	Hierarchy $\leq$ SA $<$ KK $<$ Circular $\leq$ FR
large and dense	SA $<$ KK $<$ Circular $\leq$ Hierarchy $<$ FR

Table 3.7: Algorithm Ranking With Respect to the Vertex Distribution.

algorithms of the other two layouts, while the Hierarchy algorithm is better than the circular one for small graphs, and the circular algorithm is better than the Hierarchy one for large graphs. For sparse graphs, all the algorithms produce a small number of crossings. However, the Hierarchy and the circular algorithms become less effective in terms of minimizing the number of edge crossings in dense graphs. The spring algorithms work well because they allow the vertex to be positioned at any point (with real coordinates) in the drawing plane, whereas, the circular algorithm restricts the vertices to placement onto n points along a circle, and the Hierarchy algorithm does not use the areas between layers for positioning the vertex. As the graph becomes dense, the plentifulness of the positions available for locating vertices aids the minimization of the number of edge crossings.

Remarkably, SA generally produces the smallest number of edge crossings. This also implies that SA does indeed improve the result of FR although it allows *uphill* movement—the next solution is worse than the current solution.

We can describe the ranking of algorithms in Table 3.7 according to Figure 3.18 with respect to the vertex distribution. For all the small graphs and the large graph whose $\overline{deg}$ is very small (say 2.0, which implies that the graph is very likely to be planar or even a tree), the Circular and Hierarchy algorithm are superior to the spring algorithms. When a large graph becomes dense, the KK and SA algorithm change the vertex distribution slowly, and the perfor-

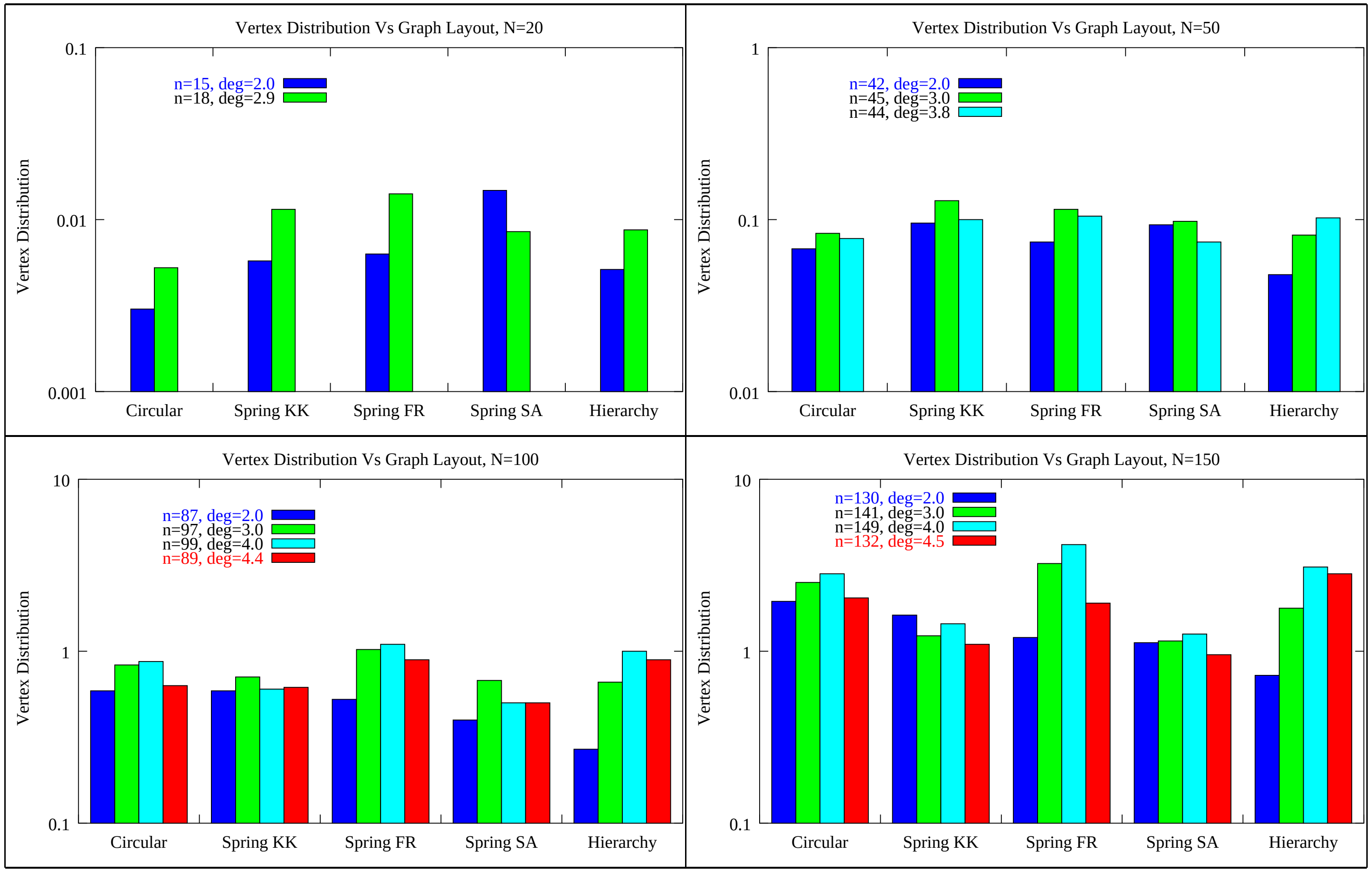


Figure 3.18: Vertex Distribution Versus Collapsed Graphs.

Collapsed Graphs	Algorithm Ranking w.r.t Run Time
small	Circular $<$ FR $\leq$ Hierarchy $\leq$ KK $<$ SA
large and sparse	Circular $<$ FR $<$ Hierarchy $<$ KK $<$ SA
large and dense	FR $\leq$ Circular $<$ KK $\leq$ Hierarchy $<$ SA

Table 3.8: Algorithm Ranking With Respect to the Run Time.

mance of the Hierarchy algorithm turns bad quickly. This is because when the graph is dense (and as a result is not like a tree), the BFS tree of the graph (see Section 2.4.2) tends to have small number of levels and thus more vertices are likely to be crowded on each level.

As well as minimizing the number of edge crossings, SA also improves the vertex distribution on the basis of the results of FR in most cases.

According to Figure 3.19, the ranking of algorithms with respect to the run time is given in Table 3.8.

On the basis of all the algorithm rankings in the table, we can abstract two partial rankings: (FR, Circular)<(KK, Hierarchy)<SA. The FR and Circular algorithm are generally much faster than the others, and SA is always the slowest. This table also suggests that the run time of the Hierarchy algorithm and SA is strongly related to the $\overline{deg}$ of the graph when N is fixed, while the other algorithms seem not to be affected too much. This is because the SA and the Hierarchy algorithm spend much time computing the number of edge crossings, and this computation becomes more time consuming as the graph becomes denser. For the graph that is likely to be planar (i.e. $\overline{deg} = 2.0$), the Circular, FR and Hierarchy algorithms run very fast.

3.5 Summary

This chapter begins with describing the representation and characteristics of the collapsed graphs. We have discussed some implementation issues of graph drawing algorithms. It has been shown that a preprocessing heuristic can improve the FR and KK algorithms implemented in the existing libraries.

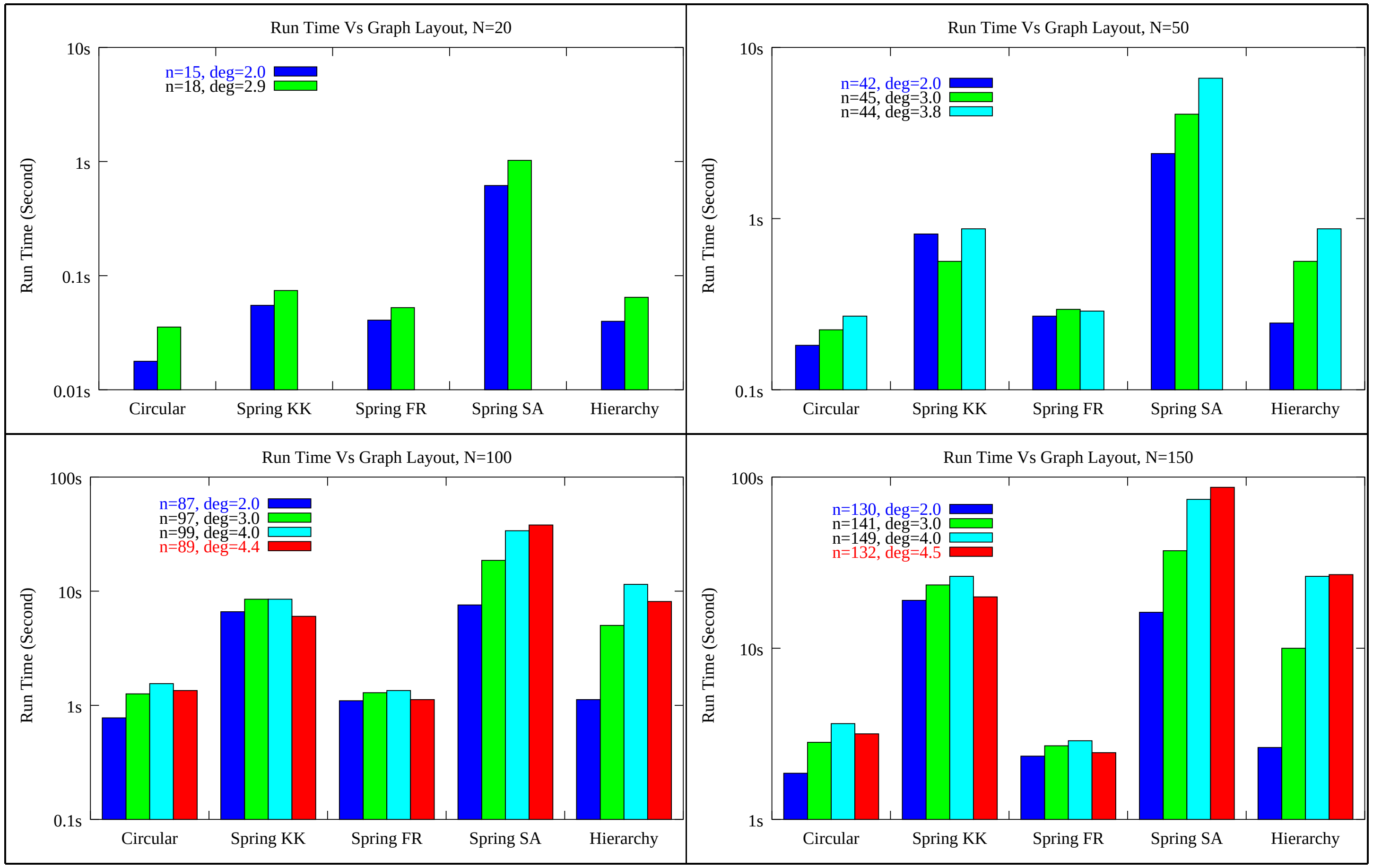


Figure 3.19: Run Time Versus Collapsed Graphs.

Collapsed Graphs	Recommended Algorithms
small and sparse	FR, Circular, KK, Hierarchy
small and dense	FR, SA
large and sparse	KK, Hierarchy
large and dense	FR, KK

Table 3.9: Recommended Algorithms for Various Types of Graphs

Since the procedure of calculating the number of edge crossings is recognized as the most time consuming part for many drawing algorithms, two efficient heuristics of counting the edge intersections are offered.

By evaluating the experimental results of various drawing algorithms, we suggest specific types of algorithms for different graphs. If the run time is a concern, Table 3.9 provides appropriate algorithms for various types of graphs.

Otherwise, the minimization of the number of edge crossings should be the major concern, and thus the SA algorithm is the best choice for all the graphs.

Chapter 4

Graph Clustering and Clustering-Based Drawing Algorithms

4.1 Introduction

Clustering [AK95][Mir96] is the process of discovering groupings or classes in data based on a chosen semantics. *Graph clustering* is partitioning a graph $G = (V, E)$ into subsets of nodes $V_1, V_2, ..., V_k$ such that $\bigcup_{i=1}^{k} V_i = V$ and $V_i \bigcap V_j = \emptyset$[1] for $i \neq j$. $V_1, V_2, ..., V_k$ are called *clusters*. Some intuitive understanding of what constitutes a cluster exists, but there is no universally accepted formal definition of "cluster". The *graph clustering problem* is to maximize or minimize $W(V_1, V_2, ..., V_k)$—the clustering criterion function of all the k clusters, where k is initially specified or automatically determined by the algorithm. The $W(V_1, V_2, ..., V_k)$ is used to quantify the clustering objectives, and it can be a function of the graph structural formulation [AK95] (e.g. the *ratio cut* introduced in Section 4.4.1.2), the geometric distance (e.g. the sum of the Euclidean distances between each node and its cluster barycenter) that is usually used for clustering a graph with geometric coordinates, or an evaluation function of the cluster assignment for each node based on given semantic qualities (e.g. the associated IP address in a subnet).

We define the clustering problem as *graph bisection* when $k = 2$ and

[1] We do not consider those cases where clusters could overlap.

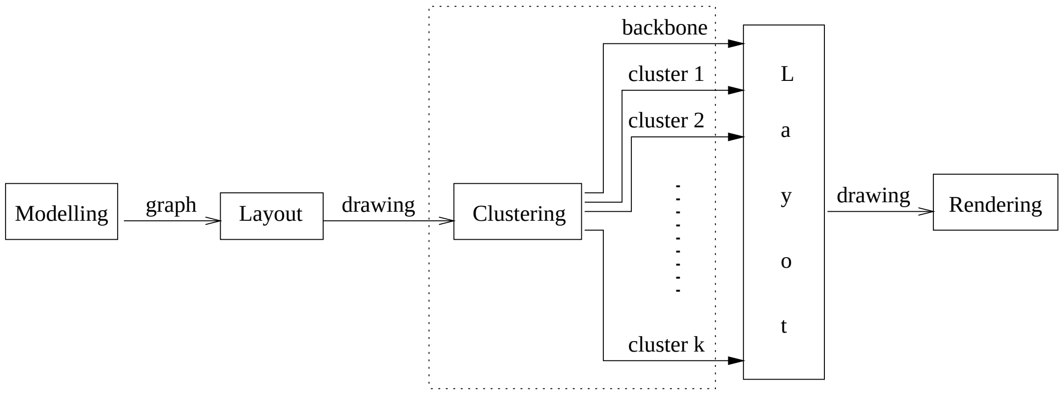


Figure 4.1: Revised Visualization Pipeline: Clustering a Graph after Graph Layout.

$|V_1| = |V_2|$; otherwise, the problem is called the *multi-way graph clustering problem*. Many applications, such as parallel computing, require both k to be a power of 2 and balanced cluster sizes, in which case *recursive bisection* (RB) is preferable. However, H.Simon and S.Teng [ST97] have shown that an ideal RB may produce a clustering $\Theta(n^2/k^2)$ times worse than a proper multi-way graph clustering. Ulrich Elsner [Els2000] collected more than 100 papers on graph bisection.

In this chapter, we will survey some auxiliary clustering approaches for graph drawing. Graph clustering can be applied either prior to or following the layout stage from the perspective of the visualization pipeline (see Figure 4.1 and 4.2). So the input of graph clustering can be either a graph without geometric coordinates or a drawing in which each vertex has been laid out by the layout stage. The process of a clustering stage will output k clusters as well as a backbone, a reduced graph used to represent the original graph. The *backbone* of G is the superstructure of its clusters $V_1, V_2, ..., V_k$: each cluster of G is represented by a vertex in the backbone. The edges of the backbone can be induced from G by the cluster connectivity or some other definition such as abridgment [HE98]. These simple graphs—clusters and the backbone—will be further positioned by a layout stage.

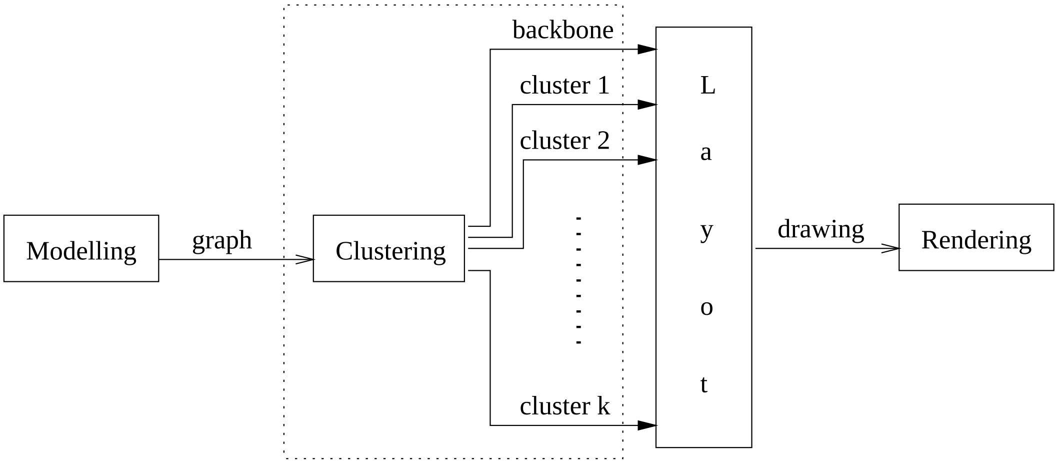


Figure 4.2: Revised Visualization Pipeline: Clustering a Graph before Graph Layout.

4.2 On Determining a Graph's Clusters and the Number of Clusters

The clusters can be determined directly if the cluster definition falls into one of the following categories: (1) a semantical classification of the node metrics; (2) a function of some graph structures, e.g. biconnectivity [DMM96]. Consequently, the number of clusters, k, is determined. Otherwise, we have two choices: select a k manually or let the algorithm find a suitable k.

If k is given by the user, iterative improvement can be used to search for a better clustering assignment on the basis of the output of the current clustering trial. If k is not constrained, a good clustering approach is *hierarchical clustering*, which clusters the graph gradually in a sequence of phases. This involves two types of heuristics: *agglomerative algorithms* and *divisive algorithms*. The agglomerative algorithm starts from n clusters (initially single vertices) to construct a smaller-sized graph by applying some cluster-merge rules, while the divisive algorithm starts from the whole graph G (i.e. 1 cluster) to construct a larger-sized graph by using some cluster-split rules. The hierarchical clustering (see Figure 4.3) runs level by level with each level producing a coarsened or uncoarsened graph for the next level and can be automatically terminated by an evaluation function of the clustering criterion development

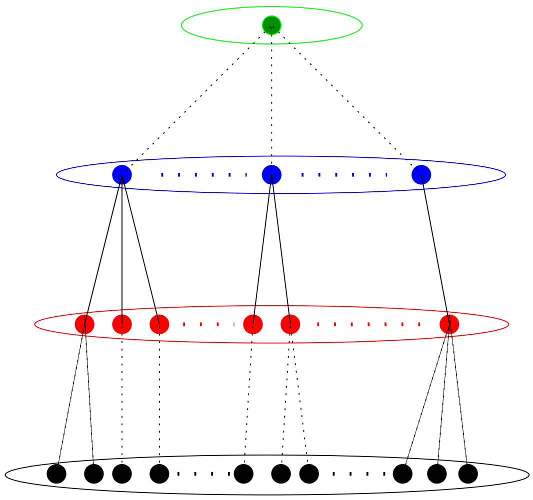


Figure 4.3: A Structure Induced by Hierarchical Clustering.

as the levels grow. When the clustering process stops, the number of clusters can be determined. Since each node has a single inheritance path throughout the clustering hierarchy, the final cluster assignments for each cluster can be consistently found.

According to its definition, after the clusters are generated, the backbone can be created correspondingly. In the following, we will describe some clustering algorithms.

4.3 Clustering a Graph Based on Geometric Information

To achieve good aesthetics, the layout approaches, in particular the force-directed approach, tend to distribute vertices evenly in the drawing area and place the connected vertices (i.e. related objects) closely (see Figure 4.4). The experiments show that the group of highly connected vertices is not likely to be fragmented after repeatedly running the layout algorithm. So it is reasonable to partition the closely positioned vertices into a cluster and then transfer the clusters and the backbone to the layout algorithm to process these simplified graphs (see Figure 4.1). In this sense, the clustering problem takes on a function of the Euclidean distance as the clustering criterion.

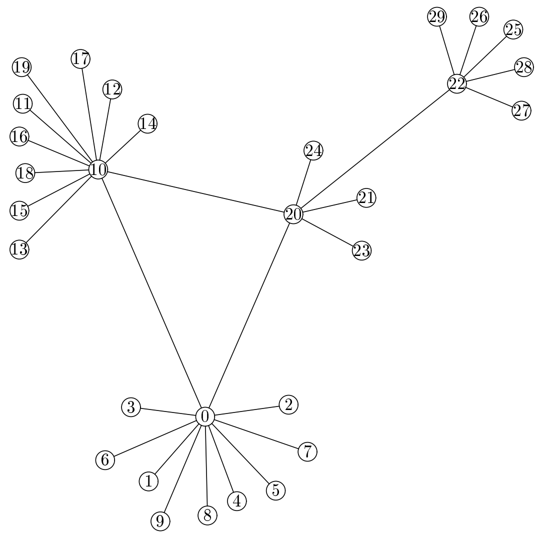


Figure 4.4: A Sample Graph Drawing.

Given a graph with geometric information (like the drawing in Figure 4.4), the clustering approaches can be generally divided into two categories: clustering with a given k and hierarchical clustering.

4.3.1 Clustering with a Given k

4.3.1.1 Bisection

The simplest geometric bisection is to find a hyperplane orthogonal for a given coordinate axis to cluster the vertices into two equal parts. For instance, if the x-coordinate is chosen to separate the nodes, we can sort the x-coordinates of the nodes and use the median value to separate the nodes.

Instead of choosing a hyperplane orthogonal for some fixed axis, the inertial bisection [HL94] searches for a hyperplane (or a line in 2D) which minimizes the sum of the squares of the distances of the nodes to the hyperplane. The method is derived from the physical model of minimal rotational inertia. For example, given a two-dimensional drawing area, the algorithm will look for a line L which goes through the barycenter $\left(\frac{1}{|V|}\sum_{v\in V} x_v, \frac{1}{|V|}\sum_{v\in V} y_v\right)$ of nodes and minimizes $\sum_{v\in V} pd^2(v, L)$ where $pd(v, L)$ is the perpendicular distance from node v to the line L.

4.3.1.2 Multi-way Clustering

The *k-means* algorithm [Ric97] works as follows: (1) it initializes k cluster means (such as barycenters) $m_1, m_2, ..., m_k$; (2)it assigns each node v to cluster V_i such that m_i has the shortest Euclidean distance to v among all the cluster means; (3) on the basis of clustering derived from step (2), it recomputes each cluster mean m_i with the coordinates of vertices in cluster V_i, and repeats step (2) until two consequent runs produce the same clustering or the iteration time reaches a given maximum.

4.3.2 Hierarchy Clustering

4.3.2.1 Agglomerative Algorithms

The original graph G is grown in a sequence of graphs $G_1, G_2, ..., G_h$ and in a bottom-up way (see Figure 4.3), where $G_1 = G$, $Cluster(G) = n$ and $Cluster(G_i) > Cluster(G_{i+1})$ for $i = 1, 2, ..., h-1$. Here, $Cluster(G)$ denotes the number of clusters in graph G. The agglomerative algorithm generates G_{i+1} on the basis of G_i by merging clusters in G_i. One typical cluster-merge heuristic is to merge the two clusters whose distance is shortest. There are two definitions for cluster distance. Given two clusters V_{ip} and V_{iq} of graph G_i, the cluster distance can be defined as $min\{d(u,v)|u \in V_{ip}, v \in V_{iq}\}$, or $max\{d(u,v)|u \in V_{ip}, v \in V_{iq}\}$ [LS99] where $d(u,v)$ denotes the Euclidean distance between u and v.

4.3.2.2 Divisive Algorithms

In contrast to the agglomerative algorithms, the divisive algorithm reduces the original graph G into a sequence of graphs $G_1, G_2, ..., G_h$ in a top-down way (see Figure 4.3). $G_1 = G$, $Cluster(G) = 1$ and $Cluster(G_i) < Cluster(G_{i+1})$ for $i = 1, 2, ..., h-1$. The divisive algorithm derives G_{i+1} from G_i by splitting some clusters in G_i. A divisive method constructs the Minimal Spanning Tree[2] (MST) of all the nodes in the drawing area and uses the MST as the unique

[2] We assume each pair of nodes has an edge whose weight is its Euclidean distance.

cluster of G to split. In this case, a good cluster-split heuristic is to remove the longest edge in the clusters [LS99].

4.4 Clustering a Graph Based on the Graph Structure

There are two criteria by which we can evaluate the quality of the clusters generated. Both metrics can be used as the clustering criteria for the structural clustering algorithms.

- *Cluster density* [AK95]: Assuming a cluster has n_i nodes and m_i intra-cluster edges, its density is $\frac{m_i}{\binom{n_i}{2}}$.
- *Cut* [RS97]: This is the sum of the cuts between all pairs of clusters. The cut $C(V_1, V_2, ..., V_k)$ is defined as

$$C(V_1, V_2, ..., V_k) = \frac{1}{2}\sum_{p=1}^{k}\sum_{u \in V_p, v \notin V_p} c_{u,v}$$

  where c_{uv} is the number (or the weight) of the edges between a vertex pair (u, v).

The structural clustering approaches can be divided into two categories: clustering with a given k and agglomerative hierarchical clustering.

4.4.1 Clustering with a Given k

4.4.1.1 Bisection

Kernighhan and Lin introduced an iterative algorithm in [KL70]. Given an initial graph clustering V_1 and V_2, the KL algorithm iteratively swaps a vertex in V_1 with another vertex in V_2 such that the cut decreases most by the switching. The algorithm proceeds in a series of passes, and in each pass every vertex is swapped exactly once, even though some pair-swaps increase the cut. The next pass takes the output of its preceding pass as the initial clustering. The algorithm stops when the pass does not produce a smaller cut than the previous pass. The KL algorithm can climb out of local minima because it allows the pair-swap that increases the cut.

4.4.1.2 Multi-way Clustering

Kernighan-Lin's two-way ratio cut algorithm [KL70] was extended to a multi-way ratio cut by T.Roxborough and A.Sen [RS97], who defined the *multi-way ratio cut* as

$$R(V_1, V_2, ..., V_k) = \frac{C(V_1, V_2, ..., V_k)}{|V_1| \times |V_2| \times .. \times |V_k|}.$$

The clustering problem becomes an attempt to split the graph into k clusters with the minimum ratio cut. The number of all the possible split ways is in the order of $\Theta(\frac{n!}{((\frac{n}{k})!)^k})$.

4.4.2 Agglomerative Hierarchy Clustering

After finding a match for a cluster, the two matched clusters can be merged into one cluster. Assuming cluster u is to be matched, the following three matching algorithms [KK99] select another cluster to match u with different considerations.

- Random maximal matching randomly selects one of the unmatched clusters adjacent to u.
- Heavy edge matching selects an unmatched adjacent cluster v such that (u, v) has the largest weight (e.g. the number of inter-cluster edges) of all the cluster pairs composed of cluster u and its adjacent clusters.
- Heavy clique matching finds an unmatched adjacent cluster v such that the merged cluster u and v has the largest cluster density of all the cluster pairs composed of cluster u and its adjacent clusters.

In [DMM96], U. Dogrusoz et al. proposed a *tree reducing* heuristic for graph clustering: the degree one node is recursively merged to its adjacent vertex until the graph has no degree one node left. The clustering algorithm will then cluster the reduced graph.

4.5 Clustering-Based Drawing Algorithms

Graph clustering can be applied either before or after the layout stage (see Section 4.1), and it can increase the performance of layout algorithms. In this section, we propose some new clustering-based drawing algorithms that involve finding the clusters of the graph and laying out the clusters.

4.5.1 Switching Clusters in a Circular Layout Drawing

The Circular algorithm of Section 3.3.1.2 rotates one vertex around a circle each time in an attempt to find the best position to minimize the number of edge crossings. As an indirect consequence, connected vertices are aggregated together. For instance, after applying the Circular algorithm, many vertices in Figure 4.5 are connected to their neighbors with adjacent orders, i.e. many edges appear on the circular circumference. Given the ordering (defined at Section 2.2) of the vertices along the circle, we denote s_i as the vertex at position i ($0 \leq i \leq n-1$). In the circular layout drawing, we define the cluster as a subset of vertices $C = \{s_{i_1}, s_{i_2}, \ldots, s_{i_l}\}$ such that $0 \leq i_1, i_2, ..., i_l \leq n-1$ and $(i_{k+1} - i_k)\, mod\, n = 1$ and $s_{i_k} s_{i_{k+1}} \in E$ for $1 \leq k < l$.

Given a circular layout drawing generated by the GSV (the Greedy algorithm that iteratively Switches Vertices. See Section 3.3.1.2), we can scan the vertex ordering to find the clusters, while minimizing the number of clusters. In Figure 4.5, 18 vertices are partitioned into 5 clusters: $\{2, 9\}$, $\{17, 12, 15, 6, 10, 16\}$, $\{1, 14, 7, 11, 0\}$, $\{5, 19, 8, 3\}$, $\{18\}$. Then we apply another greedy algorithm that iteratively switches clusters to minimize the number of edge crossings. When a cluster is chosen to be swapped with another cluster, we test not only the original ordering but also the reversed ordering of this cluster. For example, when swapping cluster $\{5, 19, 8, 3\}$, we try both $\{5, 19, 8, 3\}$ and $\{3, 8, 19, 5\}$ to find a good vertex ordering that causes the least number of edge crossings. At the end of each iteration in which a cluster has been switched with that of other clusters, if an improved vertex ordering is

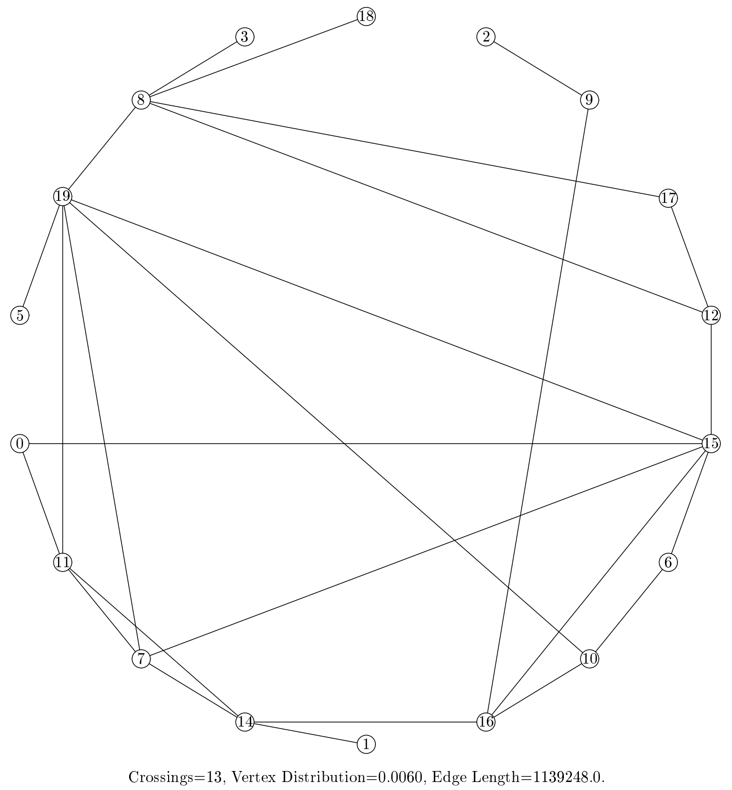


Figure 4.5: A Circular Layout Drawing Generated by the GSV.

obtained, we recompute the clusters according to the new vertex ordering.

Figure 4.6 shows a graph drawing improved by the GSC (the Greedy algorithm that iteratively Switches Clusters). Compared with the graph drawing in Figure 4.5, the one in Figure 4.6 reduces the number of edge crossings from 13 to 7.

Figure 4.7 presents the number of edge crossings reduced by the GSC on the basis of the drawing generated by the GSV. Figure 4.8 gives the ratios of the number of crossings reduced by the GSC to the original number of the GSV. It seems that the GSC is more applicable for small and sparse graphs, for which the GSC algorithm can reduce relatively more edge crossings. In contrast, the GSC reduces large number but small percentage of crossings for the large dense graphs.

For all test graphs (except the planar ones), the GDC can reduce the number of edge crossings in the drawing generated by the GSV, as indicated by Figure 4.7. This is because the GDC can move both one or more vertices

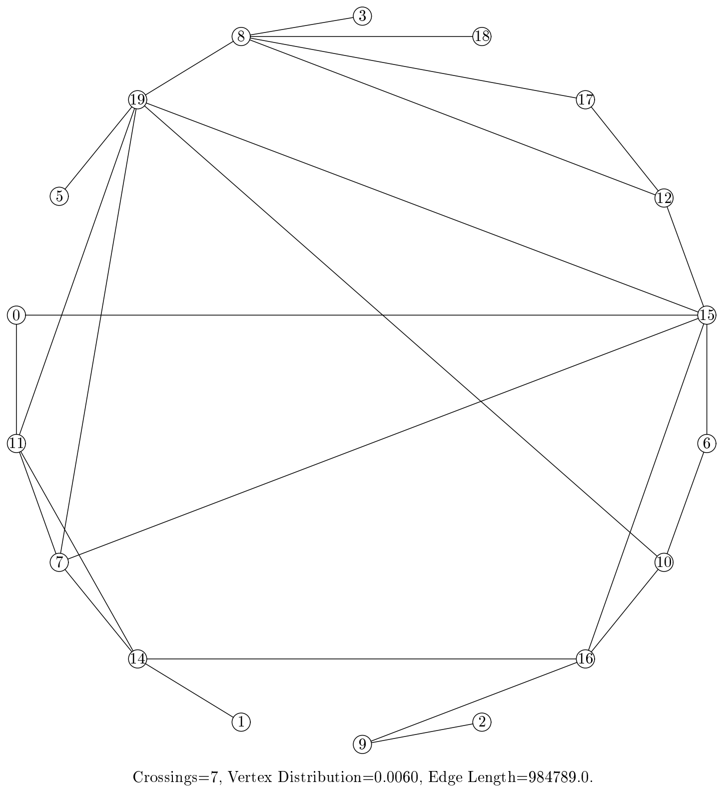


Figure 4.6: A Circular Layout Drawing Generated by the GSC.

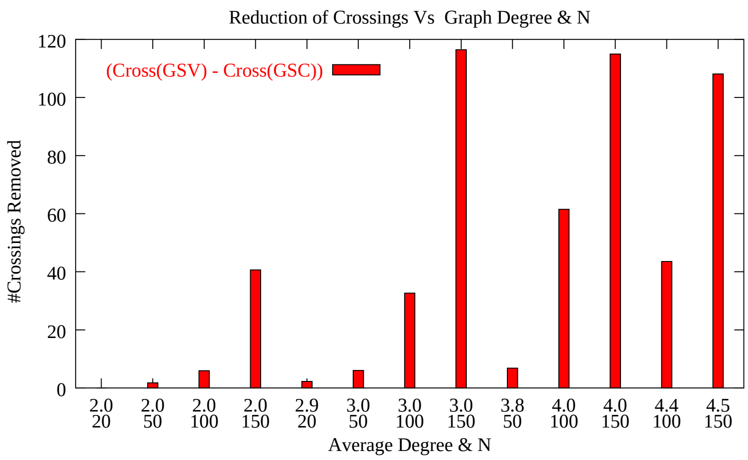


Figure 4.7: The Number of Edge Crossings Reduced by GSC Based on GSV Vs Graph $\overline{deg}$ and N.

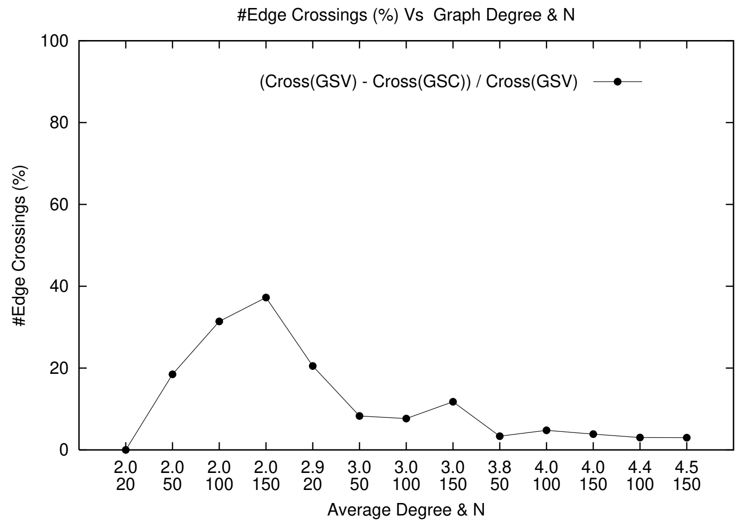


Figure 4.8: Performance of GSC over GSV Vs Graph $\overline{deg}$ and N.

(e.g. $\{2, 9\}$ in Figure 4.5), but the GSV moves only one vertex at one time. Instinctively, the GSV helps to form the clusters and to minimize the number of crossings of intra-cluster edges, and the GSC improves the drawing by minimizing the number of crossings of inter-cluster edges.

4.5.2 Zooming in on the Dense Vertex Cluster of the Spring Layout Drawing

In this section, a clustering technique is proposed to improve graph drawing generated by spring layout algorithms, which usually stabilize and yield vertex clusters that are visually apparent (see Figure 4.4). In GDC, we implement an algorithm that can detect the dense clusters in which the vetices are crowded together and then zoom in on the drawing region of this cluster and zoom out to the rest of the drawing plane. The algorithm's details are as follows.

Given a spring layout drawing (see Figure 4.9), we divide the whole drawing area into $\lceil \sqrt{n} \rceil^2$ uniform grid cells.

In most cases, the spring layout drawing produces good vertex distribution in that the number of vertices in a grid cell is small (e.g. 0, 1 or 2). However, our observation shows that there usually exists a dense cluster of grid cells in which the number of vertices is much greater than the number of cells. For

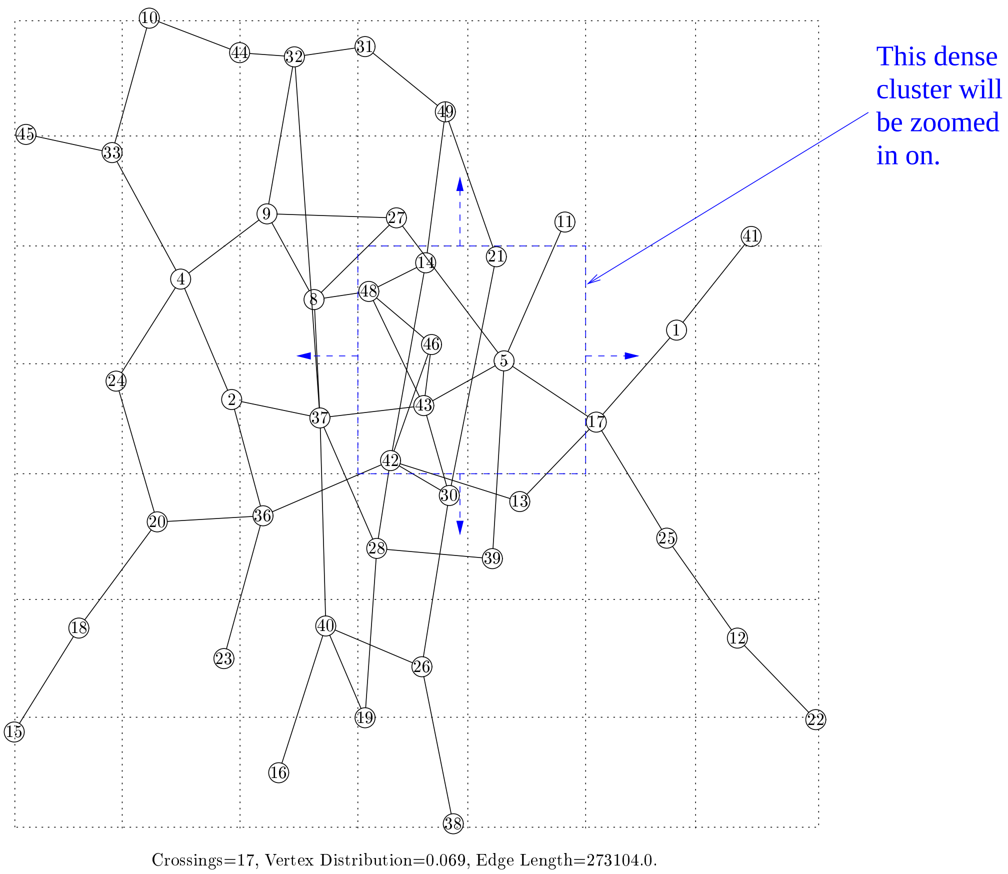


Figure 4.9: A Spring Layout Drawing Before Zooming in on the Dense Vertex Cluster

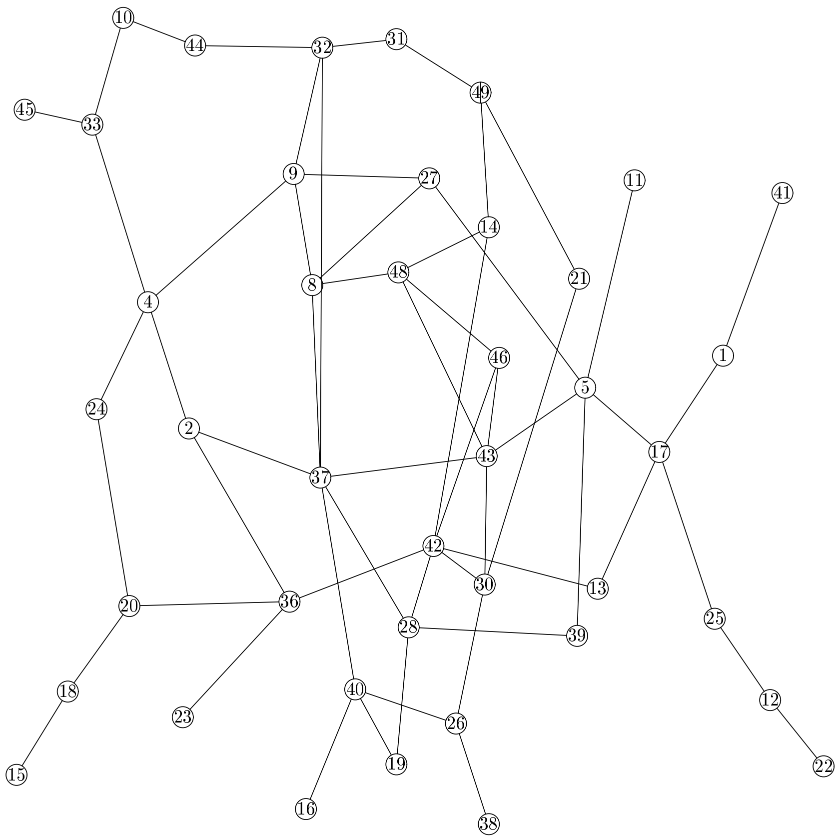
0	2	1	2	0	0	0
2	0	1	1	1	0	1
0	1	1	**3**	**2**	1	0
1	1	1	**2**	**0**	1	0
0	1	1	2	2	1	0
1	1	1	1	0	0	1
1	0	1	2	0	0	1

Table 4.1: Vertex Density Matrix of Grid Cells.

example, Figure 4.9 has 42 nodes distributed into 49 grid cells, as in Table 4.1.

We select a rectangular-shaped cluster of grid cells to zoom in on, e.g. a 2×2 rectangle. It is easy to find the dense cluster (having bold font in the Table 4.1) that hold the largest number of nodes by enumerating all the possible 2×2 rectangles in this matrix. In this case, 7 nodes are crowded within 4 grid cells. The dense vertex cluster makes the inner vertices more likely to have less geometric distance between them and hence difficult to distinguish from each other. A basic improvement of the vertex distribution is to zoom in on the rectangular region of the dense vertex cluster. When a rectangular region inside the drawing plane is magnified and the other drawing area is shrunk accordingly (see Figure 4.10), we try to preserve the graph embedding (defined in Section 2.1) and prevent the number of edge crossings from being significantly increased.

In comparison with Figure 4.9, Figure 4.10 illustrates a reduction in the vertex distribution from 0.069 to 0.058 and an increase in the number of edge crossings from 17 to 19. In Figure 4.11, we present the vertex distribution of the spring layout drawing before and after zooming in on the dense vertex cluster. For all the test graphs, the vertex distribution is improved, especially when the graph is large and dense. Figure 4.12 shows that this technique does not increase the number of edge crossings as significantly as it reduces the vertex distribution for all the test graphs.



Crossings=19, Vertex Distribution=0.058, Edge Length=364780.0.

Figure 4.10: A Spring Layout Drawing After Zooming in the Dense Vertex Cluster.

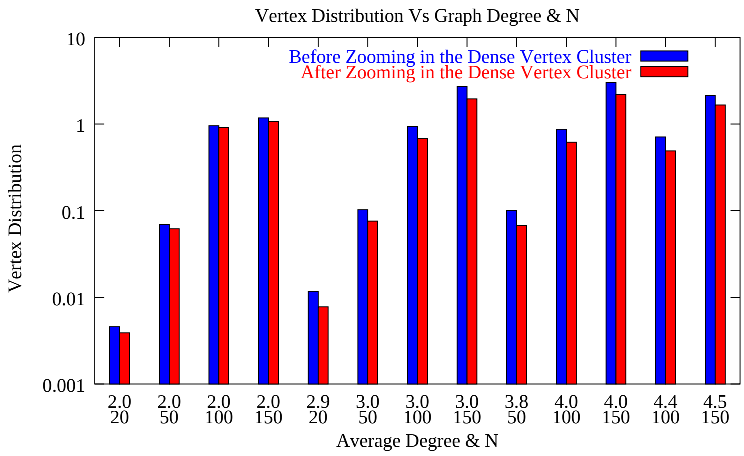


Figure 4.11: Vertex Distribution Vs Graph $\overline{deg}$ and N.

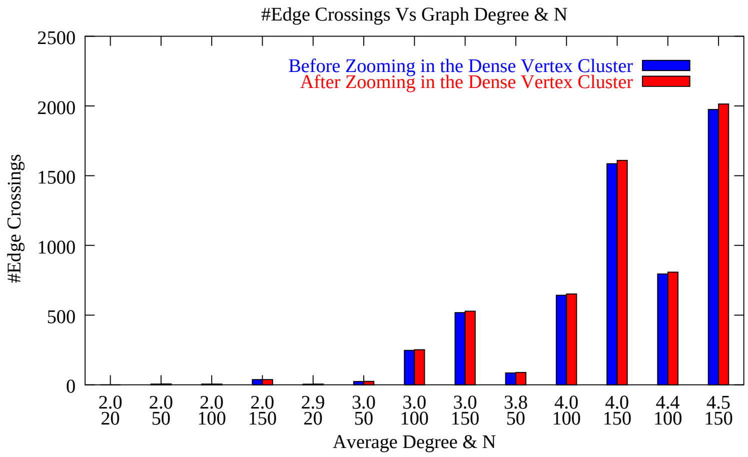


Figure 4.12: The Number of Edge Crossings Vs Graph $\overline{deg}$ and N.

4.6 Summary

In this chapter, we have shown that graph clustering is strongly correlated with graph layout, because graph clustering can simplify the input graph for drawing with the graph's geometric or structural information. We have presented a hierarchy of classifications of auxiliary clustering approaches for graph drawing. We have also developed two clustering techniques and combined them with the drawing algorithms of Chapter 3 to improve the quality of the graph drawing. The empirical results show these clustering-based drawing algorithms can significantly improve the drawing aesthetic criteria for some types of graphs.

Chapter 5

GDC System Implementation

5.1 The GDC System Architecture

The system architecture of GDC is shown in Figure 5.1, in which the components inside the rounded rectangle are I/O (flat files or memory variables) and the other components are programs (libraries or applications). The central component of GDC is two hashtables that maintain only generic information concerning the graph to be drawn. Each FDP (Frozen Development Process) uses two files to store its edge list and the information about the vertices that are frozen same (see Section 3.1.1). These files can be converted to the graph hashtables when the current edge is specified by the GUI (Graphics User Interface). The algorithm classes of GDC are written in Java, as doing so makes them platform independent, although they run less efficiently than C program. The algorithm classes interpret the input graph with the graph hashtables and store their results back to the hashtables. The algorithm library of GDC provides interfaces (whose typical arguments are the graph hashtables, the drawing method and the drawing area) to be called by any Java program.

In Figure 5.1, we see the relationship between three other components and the GDC system architecture. The GML (Graph Modeling Language) is a file format for graph representation used by some popular graph editors and layout libraries, such as Graphlet [MHB99] and VGJ [Stu96]. For any custom application, the GDC drawing library and the graph hashtables can be easily reused.

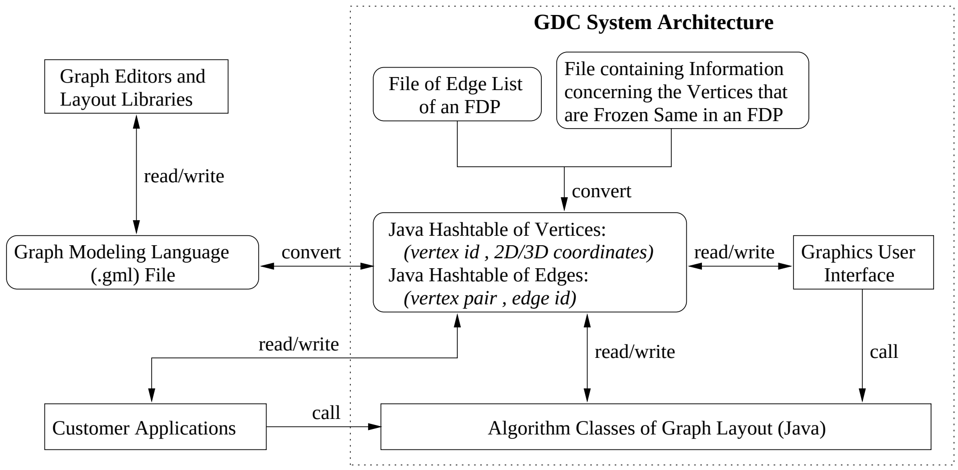


Figure 5.1: The GDC System Architecture.

5.2 Algorithm Animation

GDC can animate the process of adding edges and merging vertices throughout an FDP. As an important part of the GUI of GDC, the window of the control panel (see Figure 5.2) controls the animation.

If a file (i.e. FDP) is chosen from the file list, the information about the edges in this FDP will be filled in the "edge list", from which the user can select the current edge to view the collapsed graph. If the current edge causes some vertices to be frozen same, the information about these vertices will be displayed at the bottom right of the control panel.

Figure 5.3 demonstrates an animation of merging vertices in GDC. In this case, vertex 4 will be merged into vertex 11. To highlight the vertices to be merged, we enlarge the label sizes of vertices 4 and 11 and blink their colors as well. Then vertex 4 is pulled toward vertex 11 in a sequence of frames. Once vertex 4 has been merged into vertex 11, we increase the label size of vertex 11 and make vertex 4 gray to exhibit the effect of vertex collapse.

We employ multithreading in the implementation of the animator to provide more responsive feedback to the user. Similar to the animation of merging vertices, we implement the animation of adding edges to the collapsed graph.

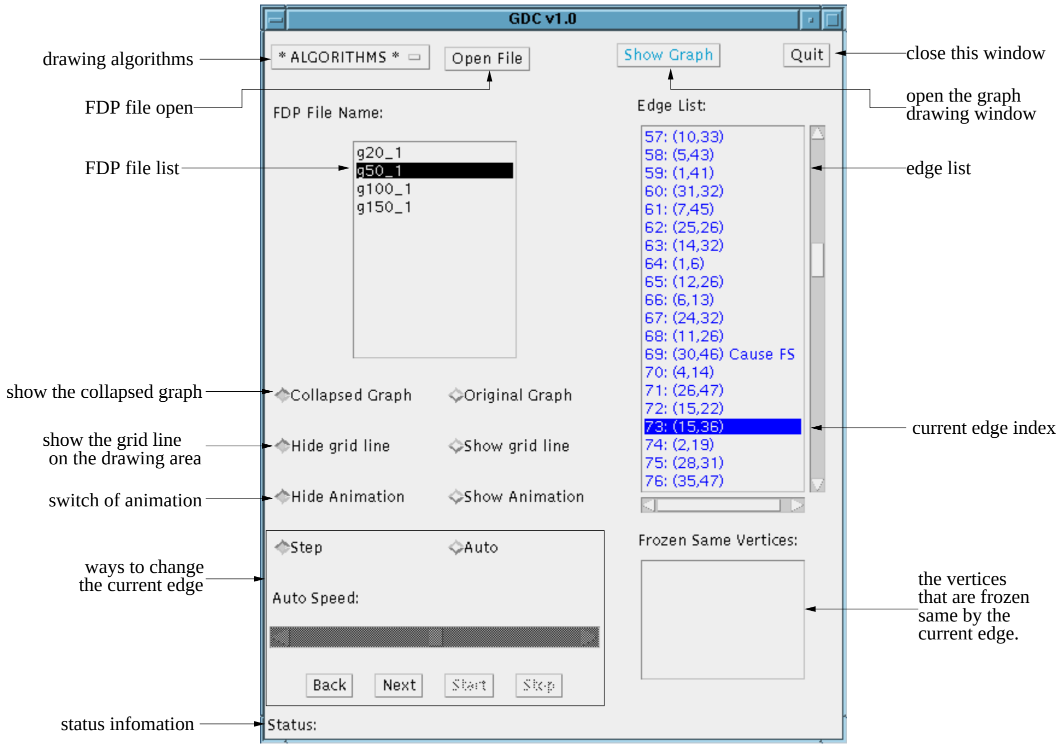


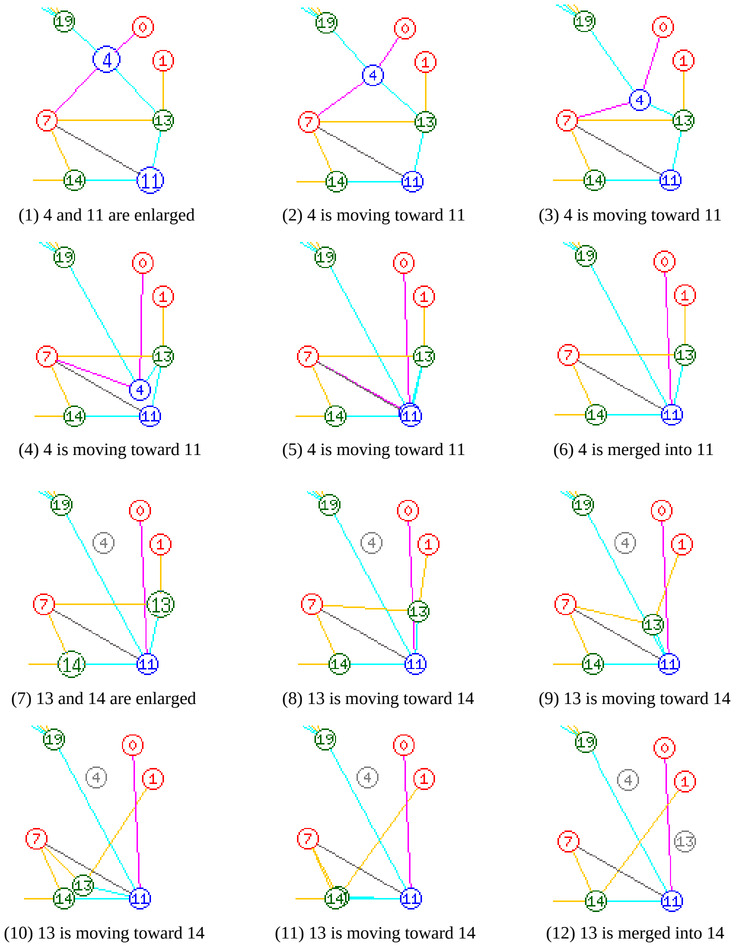
Figure 5.2: The Control Panel Window in GDC.

5.3 Three Dimensional Drawing and Visualization

We implement the 3D (three dimensional) spring algorithm based on LEDA [MN99] and the 3D visualization based on the Java 3D demos of Sun Microsystems.

In Section 2.3, we have discussed spring layout methods all of which can be applied in both a $2D$ (two dimensional) and a $3D$ drawing space. Unlike $2D$ drawing, $3D$ drawing must maintain an extra coordinate variable z. Similar to coordinate x and y, coordinate z is moved by a force function of the Euclidean distances between vertices. However, when the vertices of the input graph initially have no coordinate z, z should be initialized with scattered values in order to prevent the forces along the axis z from being zero.

Figure 5.4 presents a sample 3D drawing of the spring layout. The graph can be rotated with the mouse, providing the convenience of viewing the graph from various perspectives.



(1) 4 and 11 are enlarged
(2) 4 is moving toward 11
(3) 4 is moving toward 11
(4) 4 is moving toward 11
(5) 4 is moving toward 11
(6) 4 is merged into 11
(7) 13 and 14 are enlarged
(8) 13 is moving toward 14
(9) 13 is moving toward 14
(10) 13 is moving toward 14
(11) 13 is moving toward 14
(12) 13 is merged into 14

Figure 5.3: GDC Animation of Merging Vertices.

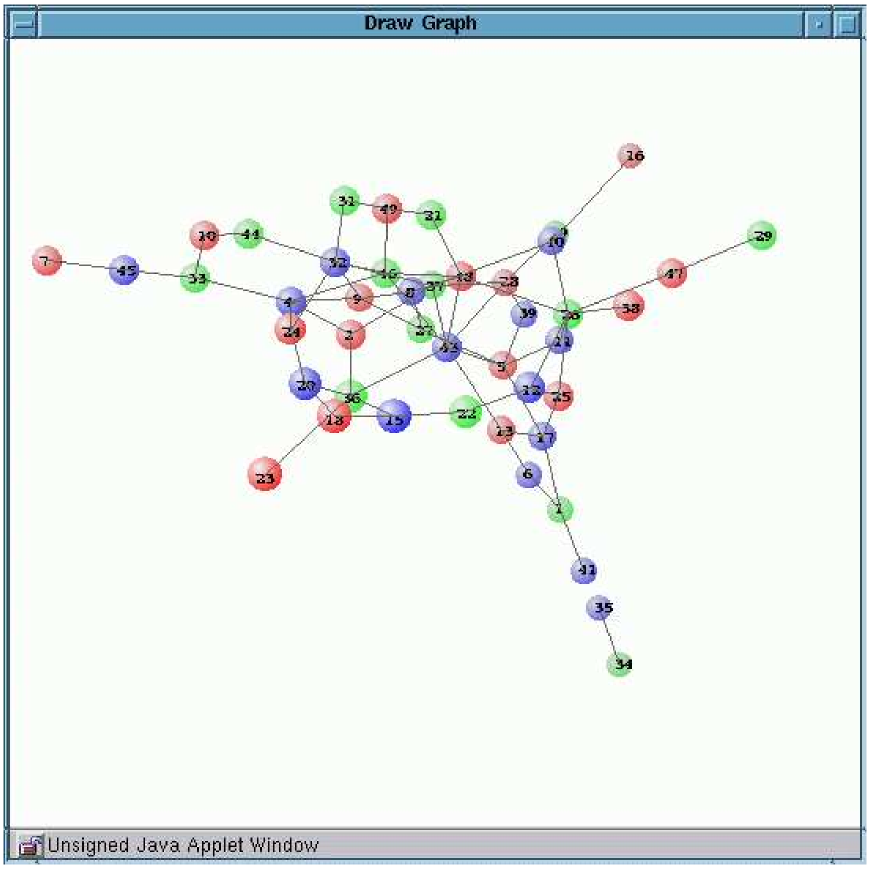


Figure 5.4: 3D Drawing of the Spring Layout.

5.4 Summary

In this chapter, we have described the system architecture of GDC and presented the relationship between the library of drawing algorithms in GDC with other applications. We have also outlined some practical concern and implementation techniques of the algorithm animation and 3D visualization provided in GDC.

Chapter 6

Conclusion and Suggestions for Future Work

6.1 Conclusion

The thesis is motivated by the desire to visualize the graph collapsing phenomenon associated with the 3-Coloring phase transition. We began by describing general graph drawing methodologies and discussing the Frozen Development Process (FDP) as well as the collapsed graph, a type of graph that represents the FDP and needs to be laid out. We selected a set of collapsed graphs to test drawing algorithms. We provided an overview of the graph drawing problem, including the drawing aesthetic criteria that can be used to measure the quality of graph drawing. We introduced the features of two famous graph drawing libraries and editors and our application—GDC.

We gave some definitions and described some problems related to the graph drawing problem. We investigated some widely used methods for three fundamental types of graph layout and attempted to represent various methods with a sequence of heuristics according to a framework for each graph layout. We empirically studied five drawing algorithms, and provided some implementation details and experimental analysis. We also presented preferable algorithms for different types of graphs. When the number of edge crossings is mostly concerned, the SA algorithm is likely to be the best.

We investigated the problem of graph clustering. We showed that graph clustering is strongly correlated with graph layout. Graph clustering can help graph drawing by reducing the input graph with geometric or structural information about the graph. We classified graph clustering methods into various categories. We also developed two clustering techniques and showed that the clustering-based drawing algorithms can significantly improve the drawing aesthetic criteria for some types of graphs.

We outlined the system architecture of GDC and proposed ways to reuse some components of GDC. We demonstrated how algorithm animation and three dimensional visualization work in GDC.

6.2 Future Work

This thesis discusses methods of graph drawing and graph clustering for drawing collapsed graphs. The techniques may be further investigated in the following ways:

- We have drawn the graph with only the straight-line drawing convention. The poly-line drawing convention needs to be investigated.

- We have applied two clustering techniques to find the clusters of a graph. Thus, we simplified the graph into small clusters of vertices for layout. More research is needed to comprehensively discover the correlation between the graph clustering and the graph drawing algorithms and visual comprehension.

Bibliography

[AK95] Charles Alpert, and Andrew Kahng. Recent direction in netlist partitioning: a survey. *INTEGRATION: the VLSI Journal*, 19:1-81, 1995.

[ASB94] D.S. Andrews, J. Snoeyink, J. Boritz, T. Chan, G. Denham, J. Harrison, and C. Zhu. Further comparison of algorithms for geometric intersection problems. In *Intel Symposium on Spatial Data Handling*, 2:709-724, 1994.

[BETT94] G.D. Battista, P. Eades, R. Tamassia, and I.G. Tollis. Algorithms for drawing graphs: an annotated bibliography. 1994. Available at http://wilma.cs.brown.edu/pub/papers/compgeo/gdbiblio.ps.Z.

[BETT99] G.D. Battista, P. Eades, R. Tamassia, and I.G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[BG2000] Sara Baase, and Allen Van Gelder. *Computer algorithms: introduction to design and analysis (3rd)*. Addison Wesley Longman, 2000.

[BHR95] Franz Brandenburg, Michael Himsolt, and Christoph Rohrer. An experimental comparison of Force-Directed and Randomized Graph Drawing Algorithms. In *Proceedings of GD '95*, pages 76-87, 1995.

[Bow98] Ivan Bowman. Methods for visual understanding of hierarchical system structures: Kozo Sugiyama, Shojiro Tagawa, Mitsuhiko Toda. Course Notes, 1998. Available at http://plg.uwaterloo.ca/~itbowman/CS746G/Notes/Sugiyama1981_MVU/index.html.

[BW97] Ulrik Brandes, and Dorothea Wagner. Random field models for graph layout. *Konstanzer Schriften in Mathematik und Informatik 33*, 1997.

[CG2000] Joseph Culberson, and Ian Gent. Frozen development in graph coloring. APES-19-2000 APES Research Report, 2000. Available at http://www.cs.ualberta.ca/~joe/Preprints/apes19.ps.

[CKT91] P. Cheeseman, B. Kanefsky, and W. Taylor. Where the really hard problems are. In *Proceedings of 12th IJCAI*, pages 331-337, 1991.

[DH91] R. Davidson, and D. Harel. Drawing graphs nicely using simulated annealing. Technical report CS89-13, Dept. of Applied Mathematics and Computer Science, the Weizmann Institute, 1989.

[DH96] R. Davidson, and D. Harel. Drawing graphs nicely using simulated annealing. *ACM Transaction on Graphics*, 15(4):301-331, 1996.

[DMM96] U. Dogrusoz, B. Madden, and P. Madden. Circular layout in the graph layout toolkit. In *Proceedings of GD '96*, pages 92-100, 1996.

[Ead84] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149-160, 1984.

[Ead92] P. Eades. Drawing free trees, *Bulletin of the Institute for Combinatorics and its Applications*, pages 10-36, 1992.

[EGS2000] Guy Even, Sudipto Guha, and Baruch Schieber. Improved approximations of crossings in graph drawings and VLSI layout areas. In *Proceedings of ACM Symposium on Theory of Computing*, 2000.

[EK86] P. Eades, and D. Kelly. Heuristics for reducing crossings in 2-layered networks. *Arts Combinatorial* 21-A:89-98, 1986.

[Els2000] Ulrich Elsner. Bibliography: graph bisection. 2000. Available at http://www-usercgi.tu-chemnitz.de/~elsner/mypartbib-e.html.

[Els97] Ulrich Elsner. Graph partitioning: a survey. Technical Report, Technische Universitat Chemnitz, 1997. Available from http://www-usercgi.tu-chemnitz.de/~elsner/index-e.html.

[EMW86] P. Eades, B. Mckay, and N. Wormald. On an edge crossing problem. In *Proceedings of the Ninth Australian Computer Science Conference*, pages 327-334, 1986.

[ESB99] Jubin Eadchery, Arunabha Sen, and Franz J. Brandenburg. Graph clustering using distance-k cliques software demonstration. In *Proceedings of GD '99*, pages 98-106, 1999.

[EW86] P. Eades, and N. Wormald. The median heuristics for drawing 2-layers networks. Technical Report 69, Department of Computer Science, Univeristy of Queensland, 1986.

[EW94] P. Eades, and S. Whitesides. Drawing graphs in two layers. *Theoretical Computer Science*, 131:361-374, 1994.

[Eve99] Guy Even. Research topics. 1999. Available at http://www.eng.tau.ac.il/~guy/research_topics.htm.

[FLM94] Arne Frick, Andreas Ludwig, and Heiko Mehldau. A fast adaptive layout algorithm for undirected graphs. In *Proceedings of GD '94*, pages 388-403, 1994.

[FR91] T. Fruchterman, and E. Reingold. Graph drawing by force-directed placement. *Journal of Software-Practice and Experience,* 21:1129-1164, 1991.

[GJ83] M.R. Garey, and D.S. Johnson. Crossing number is NP-Complete. *SIAM Journal on Algebraic and Discrete Methods*, 4:312-316, 1983.

[GKNV93] E.R. Gansner, E. Koutsofios, S.C. North, and K.P. Vo. A technique for drawing directed graphs. *IEEE Trans. Softw. Eng*, 19(3):214-230, 1993.

[HE98] M.L. Huang, and P. Eades. A fully animated interactive system for clustering and navigating huge graphs". In *Proceedings of GD '98*, pages 374-383, 1998.

[HL94] B. Hendrickson, and R. Leland. An empirical study of static load balancing algorithms. In *Proceedings of the Scalable High Performance Computer Conference*, pages 682-685, 1994.

[HRM95] Helen Purchase, Robert Cohen, and Murray James. Validating graph drawing aesthetics. In *Proceedings of GD '9*5, pages 435-446, 1995.

[HS94] D.Harel, and Meir Sardas. Randomized graph drawing with heavy-duty preprocessing. Technical report, Dept. of Applied Mathematics and Computer Science, the Weizmann Institute, 1994.

[HT74] J. Hopcroft, and R. E. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21(4):549-568, 1974.

[JM97] Michael Junger, and Petra Mutzel. 2-layer straightline crossing minimization: performance of exact and heuristic algorithms. *Journal of Graph Algorithms Application*, 1(1):1-25, 1997.

[Joh84] D.S.Johnson. The NP-Completeness column: an ongoing guide. *Journal of Algorithms*, 5:147-160, 1984.

[Kam89] T. Kamada. Visualizing abstract objects and relations: a constraint-based approach. In *World Scientific*, Singapore, 1989.

[KGV83] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science,* 220:671-680, 1983.

[KL70] B.W. Kernighan, and S. Lin. An efficient heuristic procedure for partitioning. *Bell System Technical Journal*, 49(2):291-307, 1970.

[KK89] T. Kamada, and S. Kawai. An algorithm for drawing general undirected graphs. *Journal of Information Processing Letters*, 31:7-15, 1989.

[KK99] George Karypis, and Vipin Kumar. A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1), 1999.

[LS99] Anatoly Likhatchev, and Costa Siourbas. Cluster aanalysis: determining the number of clusters. Course Notes, 1999. Available at http://www.cs.mcgill.ca/~tolik/CS-644/cluster.html.

[Mak88] E. Makinen. On circular layouts. *International Journal of Computer Mathematics*, 24:29-37, 1988.

[Man90] J.B. Manning. Geometric symmetry in graphs. PhD thesis, Purdue University, 1990.

[MHB99] Petra Mutzel, Michael Himsolt, and Franz J. Brandenburg. Design, analysis, implementation, and evaluation of graph drawing algorithms. 1999. Available at http://www.infosun.fmi.uni-passau.de/Graphlet/screen.gif.

[Mir96] B. Mirkin. *Mathematical Classification and Clustering.* Kluwer Academic Publishers, 1996.

[MKNF87] S. Masuda, T. Kashiwabara, K. Nakajima, and T. Fujisawa. On the NP-Completeness of a computer network layout problem. In *Proceedings of IEEE 1987 International Symposium On Circuits and Systems*, pages 292-295, 1987.

[MM96] K. Mehlhorn, and P. Mutzel. On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. *Journal of Algorithmica*, 16:233-242, 1996.

[MN99] K. Mehlhorn, and St. Nher. *The LEDA Platform of Combinatorial and Geometric Computing.* Cambridge University Press, 1999.

[Ric97] Duda Richard. Feature selection and clustering for HCI. Technical Report, Department of Electrical Engineering, San Jose State University. 1997. Available at http://www-engr.sjsu.edu/~knapp/HCIRDFSC/FSC_home.htm.

[RS97] T. Roxborough and A. Sen. Graph clustering using multiway ratio cut. In *Proceedings of GD '97*, pages 291-296, 1997.

[RT81] E.M. Reingold, and J.S. Tilford. Tidier drawing of trees. *IEEE Transactions on Software Engineering*, 7(2):223-228, 1981.

[ST97] H. Simon, and S.H. Teng. How good is recursive bisection. *SIAM Journal on Scientific Computing*, 18(5):1436-1445, 1997.

[ST99] Janet Six, and Ioannis Tolis. A framework for circular drawings of networks. In *Proceedings of GD '99*, pages 107-116, 1999.

[STT81] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Trans. Sys., Man, and Cybernetics*, 11(2):109-125, 1981.

[Stu96] Shawn Stutzman. Java graph drawing tool. 1996. Available from http://www.eng.auburn.edu/department/cse/research/graph_drawing/graph_drawing.html.

[Tom2000] Tom Sawyer Company. Graph layout toolkit product literature. 2000. Available from http://www.tomsawyer.com/get/get-windows.html.

[Tsa93] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[Tun94] Daniel Tunkelang. A practical approach to drawing undirected graphs. Master's thesis, CMU, 1994.

[Tut63] W.T. Tutte. How to draw a graph. In *Proceedings of London Mathematical Society*, 13(3):743-768, 1963.

[War77] J.N. Warfield. Crossing theory and hierarchy mapping. *IEEE Trans. Syst. Man, Cybern.*, 7(7):502-523, 1977.