An Improved Spring-based Graph Embedding Algorithm and LayoutShow: a Java Environment for Graph Drawing

Lila Behzadi

A thesis submitted to the Faculty
of Graduate Studies in partial fulfillment of the
requirements for the degree of

Master of Science

Thesis Supervisor: Professor Joseph Liu

Graduate Programme in Computer Science
York University
4700 Keele Street, North York, Ontario
M3J 1P3, Canada

Abstract

Algorithms based on force-directed placement and virtual physical models have become one of the most effective techniques for drawing undirected graphs. Spring-based algorithms that are the subject of this thesis are one type of force-directed algorithms. Spring algorithms are simple. They produce graphs with approximately uniform edge lengths, distribute nodes reasonably well, and preserve graph symmetries. A problem with these algorithms is that depending on their initial layout, it is possible that they find undesirable drawings associated with some local minimum criteria. In addition, it has always been a challenge to determine when a layout is stable in order to stop the algorithm.

In this thesis, we develop a simple but effective cost function that can determine a node layout quality as well as the quality of the entire graph layout during the execution of a Spring algorithm. We use this cost function for producing the initial layout of the algorithm, for helping nodes move out of unwanted local minima, and for providing robust stopping criteria. Furthermore, as a part of this thesis we develop *LayoutShow*: a signed applet/application for graph drawing and experimentation which includes the implementation of our Spring algorithm which we call *CostSpring*.

Acknowledgement

First and foremost, I would like to thank my thesis supervisor, Joseph Liu, for his valuable comments and ideas, for our long discussions, his encouragements, and our group lunches on Tuesdays. I would also like to thank Andy Mirzaian, Eshrat Arjomandi, and Asia Weiss, the members of my examining committee, for taking the time to read my thesis and attending my oral examination.

NSERC provided financial support through a scholarship, for which I am truly grateful.

Many thanks to Arne Frick, Daniel Tunkelang, and Michael Himsolt for clarifying many concepts of force-directed graph drawing algorithms and their implementations for me throughout the course of my thesis.

I thank my parents for believing in me, for being my role-models for hard-work and dedication, and for their tremendous support and understanding during my schooling years. Many thanks should also go to my sister and my brother in-law for providing me with a place during the last two years of my undergraduate studies which made it possible for me to pursue graduate school. I would like to thank Mike for his endless support and understanding, for helping me use JClass chart, storing back-up copies of my thesis, and just being there when I most needed.

At last, I thank my friends at York university, particularly, Sanaz Rastegari-rad, Hala Sroujian, Saeed Hossani, Laura Apostoloiu, Ovidiu Miclea, and Matthew Izatt for making my undergraduate and graduate years at this university so much more enjoyable.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As information in the form of entities and relationships grow and become more complicated, visualization can assist in their understanding. When entities and their relationships are respectively stored as the nodes and edges of a graph, the problem of their visualization becomes the problem of graph drawing. Computer-automated graph drawing algorithms input the adjacency information of a graph and output the positions of graph nodes that would present a pleasing drawing of the graph. In recent years, such algorithms have become more popular. This is primarily due to the difficulties of drawing graphs, with perhaps more than 10 nodes, by hand or by graph editors. In addition, the availability of libraries for creating graphic user interfaces, and more powerful computer graphics hardware have taken away some of the obstacles on the way to computer-automated graph drawing.

The variety in the nature of the entities and relationships who need visualization has sub-divided the graph drawing field into areas such as: straight line drawing of undirected graphs, hierarchical drawing of directed graphs, or orthogonal drawing. Force-directed spring-based algorithms, which have been developed for drawing

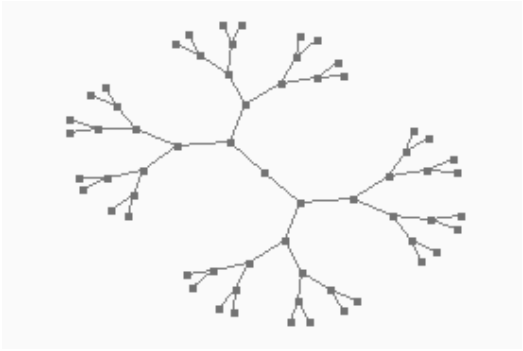

Figure 1.1: Binary Tree $|V|$=63 $|E|$=62

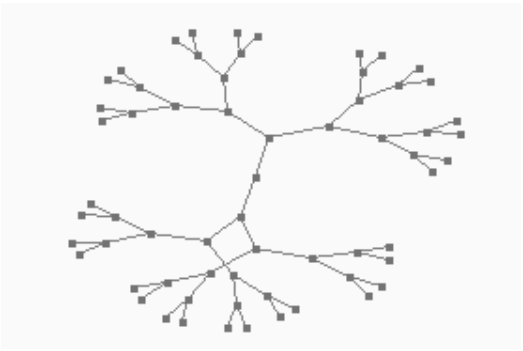

Figure 1.2: Binary Tree $|V|$=63 $|E|$=62

straight-line undirected graphs, are the subject of this thesis.

Figures 1.1 and 1.2 show two layouts of a complete binary tree with 63 nodes which have been generated by a spring-based algorithm. The layout of Figure 1.2 is in an undesirable local minimum. Examples of a variety of graph layouts produced by a spring-based algorithm can also be found in Appendix C.

1.1 Motivation

A *graph* G consists of a set E of *edges* and a set V of *vertices*. An edge of a graph is a pair (u, v) where $u, v \in V$. If the pair is not ordered the graph is *undirected*. In an undirected graph, a pair $(u, v) \in E$ implies that u is adjacent to v and v is adjacent to u [41]. Graph drawing algorithms find the locations of the vertices of a given graph to generate a pleasing drawing of the graph according to certain aesthetic criteria.

In Spring algorithms, nodes of a graph are viewed as charged particles which impose repulsive forces on one another, and the edges of the graph are viewed as springs causing attractive forces between adjacent nodes of the graph. Spring algorithms in each iteration compute the forces on each node and displace the nodes according to

the overall forces acting on them. In effect, these algorithms seek a drawing of a given graph for which the forces on each node is locally minimum.

Spring algorithms may reach at an unpleasing local minimum depending on their initial layout. In addition, although the forces indicate the direction of movement for a node, often, their overall magnitude is too large and it has to be limited for the algorithm to force it to converge. The amount by which the forces are limited is called the *temperature*. Determining the temperature of the nodes in each iteration of a Spring algorithm is called *temperature scheduling*. In addition to initial layout, temperature scheduling is also a determining factor in whether a graph layout ends up in an unpleasing local minimum or not. Finally, once a graph layout cannot significantly improve, the algorithm should terminate.

Therefore, the challenges that currently Spring algorithms face include: the initial placement of nodes, temperature scheduling, and the algorithm termination criterion.

In addition, even though the existing graph drawing softwares often include Spring algorithms, they do not support features that would facilitate experimentation with these algorithms. Some of these features include: running-time of an algorithm execution, the number of iterations, node-based and iteration-based animations, and a variety of different initial graph layout algorithms.

1.2 Objectives

The objective of this thesis is to improve on the existing Spring algorithms in the following areas:

- Reducing the chances of finding graph layouts that are in undesirable local minima.
- Finding a robust method for terminating a Spring algorithm once its layout has converged.

Furthermore, we intend to develop a software that would facilitate experimentation with Spring algorithms.

1.3 Contributions

We have developed an improved Spring algorithm, *CostSpring*, which makes use of a cost function for determining the quality of a node and a graph layout. The node qualities are used in temperature scheduling. The graph layout quality is used in terminating the algorithm as well as in an initial layout algorithm which we call *CostChosen*. This initial layout algorithm aims at reducing the chances of finding graph layouts that are in unwanted local minima.

Furthermore, we implement *LayoutShow*: a signed applet/application for graph drawing and experimentation [5]. The followings are some of the main features of this software:

- Variety of Spring algorithms: FR [16], KK [21], GEM [15], and CostSpring.
- Initial layout algorithms: randomized, Insert [15], and CostChosen.
- Node-based and iteration-based animations.
- Running-time and the number of iterations of a spring algorithm.

- GML [19], a widely used file format for both graph and layout specifications.
- A signed applet to allow file I/O on the local machine.

1.4 Outline of the Thesis

The organization of this thesis is as follows: in Chapter 2, we review the background and related work in the area of force-directed graph embedding algorithms. Chapter 3 describes our approach to improve on the existing Spring algorithms. Chapter 4 covers the design and implementation of *LayoutShow*. Chapter 5 presents the experimental results of comparing the running-time and quality of our algorithm with existing spring-based algorithms. Lastly, Chapter 6 presents the conclusion of this thesis and possible avenues for future work.

Chapter 2

Background

In this Chapter, we present the work related to this thesis in the area of force-directed methods for graph layout and Java-based [34] graph drawing software. The bibliography of the work done in the field of computer automated graph drawing can be found in [3] and [4]. We start by stating the approach of a force-directed method for graph drawing, and describe two types of force-directed methods: the Spring and Electrical force-based and Barycenter methods. We continue by introducing the various force-directed algorithms for graph drawing. Then, we present the results of an experimental comparison of these algorithms given in [7]. Finally, we review the existing Java-based environments for graph drawing.

2.1 Force-directed Methods

These methods aim at drawing general undirected graphs which is the focus of this thesis. Furthermore, there have been some variations of this method developed for drawing directed and hierarchical graphs [32], [37]. Generally, force-directed methods

view a graph as a system of particles with forces acting between them. Each graph node is considered to be a particle, and the goal is to position the graph nodes so that each node has locally minimal energy. In other words, force-directed methods seek for an equilibrium state of the force system where the sum of the forces acting on each particle is zero. As defined in [4], force-directed methods consist of two parts: a *model* and an *algorithm*. The model is a *force system* defined by vertices and edges so that the equilibrium state of the system is a pleasing drawing. On the other hand, the algorithm is a technique for finding the equilibrium state of the nodes and edges for a given graph.

2.1.1 Springs and Electrical Forces

One of the simplest and most popular force-directed methods uses *spring* and *electrical* forces. The algorithms that use this form of forces are called *Spring* algorithms. The edges of the graph are viewed as springs that connect the nodes that are considered to be charged particles. In this method, the force on a node v is defined as follows

$$F(v) = \sum_{(u,v)\in E} f_a(d_{uv}) * (\frac{u-v}{d_{uv}}) - \sum_{(u,v)\in V^2} f_r(d_{uv}) * (\frac{u-v}{d_{uv}}) \tag{2.1}$$

where d_{uv} is the distance between u and v, and f_a (the attractive factor) and f_r (the repulsive factor) are based on Hooke's law and electrical repulsion respectively which are defined as follows

$$f_a(d_{uv}) = k_{uv}^{(1)}(d_{uv} - l_{uv}) \tag{2.2}$$

$$f_r(d_{uv}) = \frac{k_{uv}^{(2)}}{d_{uv}^2} \tag{2.3}$$

where l_{uv} is the natural length of the spring between u and v, $k_{uv}^{(1)}$ is the *stiffness* of the spring between u and v, and $k_{uv}^{(2)}$ indicates the strength of the repulsive force between u and v.

The spring force between u and v forces the distance between these two vertices to become equal to l_{uv} which is normally equal to a constant value for all edges. In addition, the repulsive electrical forces prevent the nodes of the graph to be placed too close to one another.

As we will see in Section 2.2, various Spring algorithms use attractive and repulsive factors (i.e., f_a and f_r) other than what have been introduced in this Section.

2.1.2 Barycenter Method

One of the earlier works in the area of graph drawing was introduced by Tutte [38], [39]. In his method which is similar to the method described in Section 2.1.1, there is no electrical charge, the stiffness of each spring is set to one, and the length of each spring is set to zero. Therefore, the force acting on a vertex v is given by

$$F(v) = \sum_{(u,v)\in E} (u - v) \tag{2.4}$$

Equation 2.4 results in an undesirable equilibrium state where all the nodes are located at the origin. To solve this, a set of at least three vertices are fixed at the corners of a convex polygon. The rest of the vertices are free and originally located

at the origin. The algorithm iterates and in each iteration a new location (x_v, y_v) is computed for each free vertex v using equation 2.4 and the current positions of the remaining nodes until x_v and y_v converge for all free vertices.

2.2 Various Force-directed Algorithms

2.2.1 Eades' Algorithm

Eades introduced the first Spring algorithm for graph drawing with electrical forces [14]. In his scheme, he used logarithmic strength springs in place of Hooke's law, and inverse square law for repulsive forces. In addition, he only made non-adjacent vertices repel each other. The following is Eades's algorithm as stated in [14].

```
algorithm SPRING(G:graph)
   place vertices of G in random locations;
   repeat M times
     calculate the force on each vertex;
     move the vertex C4*(force on vertex)
   end \\ repeat
   draw graph
 end \\ SPRING
```

where C4=0.1, and M=100. Eades' algorithm has been the basis for the subsequent Spring algorithms. The major drawbacks of this algorithm are the high cost of computing the attractive and repulsive factors, and the absence of repulsive forces between adjacent nodes which can result in concentration of a number of adjacent nodes in

a small area. In addition, this algorithm uses simple methods for stopping and for choosing the amount of displacement of nodes which are non-adaptable to the graph and node qualities. The Spring algorithms that came after this aimed at making improvements in these areas. These algorithms are described below.

Note that in this thesis, one iteration of the repeat-loop in the Eades' algorithm is referred to as one *global iteration*. The same outer loop exists in the later improved algorithms.

2.2.2 FR

Fruchterman and Reingold extended Eades' work by introducing new attractive and repulsive factors as well as a temperature and a cooling schedule in an algorithm that is known as FR [16]. In this algorithm, the force acting on a vertex v is computed as stated in Equation 2.1. However, FR uses the following attractive and repulsive factors which they show can produce results similar to Eades' layouts but are more efficient to compute

$$f_a(d_{uv}) = \frac{d_{uv}^2}{k} \tag{2.5}$$

$$f_r(d_{uv}) = \frac{-k^2}{d_{uv}} \tag{2.6}$$

Within each iteration of FR, forces on all nodes are computed, and then all the nodes are moved at once. This algorithm limits the maximum displacement to a temperature t where t is computed by a cooling function for all nodes at the end of each iteration. They suggest a temperature that starts at an initial value and decreases in

an inverse linear fashion. However, they also note that a better cooling schedule can dramatically change resulting layouts and reduce the number of iterations required to find the layout. To speed up their algorithm they use what they call *grid-variant algorithm*. In this variant the drawing area is divided into grids. The attractive forces are computed as usual, but for each vertex the repulsive force is only computed between the vertex and the nodes in its close-by cells of the grid. This method has not been very popular mainly due to its dependency on the distribution of nodes within the grid, the overhead of placing nodes in grid cells in each iteration, and the compromised layout quality that it produces in some cases.

FR terminates after a maximum number of iterations is exhausted. Since different types of graphs require different number of iterations for their layouts to converge, this method of stopping may terminate too early or too late depending on the graph. Graphlet [20] version of FR terminates when the maximum overall force magnitude of all nodes falls below 3 or if the maximum number of iterations is reached. As our experiment in Section 5.3.3 shows this method of stopping works only for small graphs with less than 50 nodes. In addition, FR makes use of a drawing frame that limits the nodes' displacements. However, most algorithms including the Graphlet version of FR do not add this extra restriction on the displacement of nodes, and use rescaling to draw the final layout in a frame of a fixed size.

2.2.3 GEM

GEM [15], a short form for *graph embedder*, is a Spring algorithm based on Eades' algorithm, and is the first one that uses the history of a node's movement to choose

the temperature for the current displacement of the node. Unlike FR, this algorithm moves the nodes one at a time in each iteration. The following is an outline of this algorithm:

```
algorithm GEM(G:graph)
  while T_global > T_min and #round < R_max do
    find a random permutation of vertices;
    for each vertex v from the random permutation
       v's impulse = attraction to center of gravity +
                     random disturbance +
                     repulsive forces + attractive forces;
       update v's position limited by its individual temperature;
       update v's temperature;
    end //for
  end // while
end // GEM
```

where `T_min` and `R_max` are constants that indicate the minimum global temperature and the maximum number of rounds respectively. `T_global` is defined as the average of the local temperatures of all vertices. The local temperature for each vertex is computed based on the old local temperature and the likelihood that the vertex is oscillating or rotating. GEM determines whether a node is oscillating, rotating, or moving towards its final destination based on the directions of movements in the previous displacements of the node. In case of oscillation or rotation, the local

temperature of the node is decreased, and otherwise is raised. Our experiments of Section 5.3.3 show that for all of our test graphs GEM fails to stop before `R_max` is reached. In addition starting from random graph layouts, GEM finds drawings with many edge crossings for large planar graphs. We must note that GEM introduces an initial layout algorithm named *insert* that inserts the vertices one-by-one in an initial round. The layouts of some types of graphs such as binary trees and grids generated by GEM may benefit from an initial layout produced by this insert algorithm.

2.2.4 Modeling Graph Theoretic Distances: KK

Kamada and Kawai introduced an algorithm based on graph-*Euclidean* and graph-*theoretic* distances between pairs of vertices[1], which has been referred to as the KK algorithm [21]. In this algorithm, the graph nodes are considered as particles that are all connected by springs whose ideal lengths are equal to the graph-theoretic distances between their two end-point particles multiplied by the desirable length of one edge. The goal of the KK algorithm is to find a balanced spring system. The energy of KK's spring system is defined by

$$E = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} k_{v_i v_j} (d_{v_i v_j} - l_{v_i v_j})^2 \tag{2.7}$$

where $l_{v_i v_j}$ corresponds to the desirable distance between v_i and v_j, and $k_{v_i v_j}$ is the strength of the spring between these two vertices. KK computes a local minimum of E and it solves the minimization problem using the Newton-Raphson method.

[1] The graph theoretic distance between the nodes v_i and v_j in a graph G is the length of a shortest path between these two nodes in the graph.

As a result, they define E_i, the energy of a node v_i, using the partial derivatives of Equation 2.7 by x_i and y_i. The following is an outline of this algorithm

```
algorithm KK(G:graph)
  while max(E_i for 1<= i <=n) > epsilon do
    let v_m be the particle satisfying E_m = max(E_i for 1<= i <=n)
    while E_m > epsilon do
      compute displacement of v_m by solving a system of partial
      differential equations
      displace v_m
    end
  end
end //KK
```

where n is the number of nodes. The major drawback of this method is its high computational cost due to the work in solving a system of partial differential equations in the inner while-loop of the algorithm. Graphlet [20] version of KK uses a maximum of $10 \times n$ iterations of the inner loop. Although, this modification speeds up the algorithm and generates good quality layouts for small graphs, for large graphs it generates layout of planar graphs with many edge crossings.

2.2.5 Tunkelang

Tunkelang has presented two different algorithms for graph drawing [36], [37]. In this section we outline these two approaches.

A Practical Approach to Drawing Undirected Graphs

The first algorithm introduced by Tunkelang [36] is an incremental algorithm which is often referred to as Tu. This algorithm has three stages. In the first stage, a permutation of the nodes of a given graph is constructed from a minimal height breadth-first spanning tree of the graph. In the second stage, for each node in the order computed in the first stage, the area surrounding the already positioned neighbors of the node is sampled and examined for an approximate best position, and the node is placed at this position. The cost function to determine a best location is that of DH explained in Section 2.2.6. In the second stage, when the local optimization procedure improves a vertex's position, it recursively performs the process on the already placed adjacent nodes of the vertex. Finally in the last stage, the algorithm performs the local optimization process at every node for fine-tuning.

This algorithm generates layouts that are different from those of FR, KK, and GEM [7]. Specifically, it does not capture graph symmetries. In addition, the algorithm takes a quality parameter where a large value results in better quality of graph layouts. However as reported by Himsolt et al. [7], this quality parameter is estimated to have an exponential effect on the running-time of the algorithm.

Efficient Computation of Forces and The Optimization Process

The second graph drawing algorithm produced by Tunkelang [37] is based on the FR algorithm. This algorithm differs from FR in its computation of repulsive forces, in its optimization process, and in stopping the algorithm. Similar to the "grid

variant" in FR, this algorithm approximates the computation of repulsive forces on a node. Here, this is done using a Barnes-Hut tree-code [2]. FR's optimization process uses force laws that in effect compute the negative gradient of an implicit objective function. However, this algorithm uses the conjugate gradient method on a non-quadratic objective function using an approximate line search (see e.g. [17], for optimization techniques). In addition, this algorithm uses the average of the square of the displacement distances of nodes for stopping the algorithm. It terminates when this value is less than 0.01.

We must note that there are not enough details in [37], particularly, in explaining the optimization process, the algorithm's source code is not available to the public, and the Java applet JIGGLE (whose Internet address can be found in the same article) currently does not work. As a result, it is difficult to compare the performance and the quality of the layouts of this algorithm with those of the other algorithms. However the time measurements by Tunkelang on square grids, complete binary trees, and hypercubes show that the optimization process speeds up the algorithm in comparison with FR. On the other hand, the overhead of Barnes-Hut approximation of the repulsive forces slows down the algorithm for graphs with approximately less than 150 nodes, and speeds up the algorithm for graphs larger than that. In addition, he reports that the method of stopping the algorithm is a conservative one and in many cases the algorithm could be stopped much sooner.

2.2.6 Simulated Annealing: DH

Davidson and Harel [12] use the simulated annealing approach [22] to graph drawing. Their algorithm is often referred to as DH. In the context of simulated annealing for graph drawing, a *configuration* is the assignment of unique grid points to nodes of a given graph. The algorithm tries to find an optimal configuration according to a cost function. In DH, the rule for generating a new configuration is by moving one node to a randomly picked point on a circle around the vertex with radius r, only if the new configuration is a better one according to a cost function described below. Radius r is initially large and decreases in each round of the algorithm. This decreasing radius (i.e. the decreasing neighborhood in simulated annealing terms) makes this algorithm different from the simulated annealing process in which a constant rule for generating new configurations is used. For a layout L of graph G, the cost function of DH as defined in [4] and [12] is

$$f(L) = \sum_{i=1}^{5} \lambda_i f_i(L), \tag{2.8}$$

where λ_i is the weight of $f_i(L)$, and

$f_1(L) = \sum_{(u,v)\in V}(1/d_{uv}^2)$ where d_{uv} is the distance between u and v. This is to prevent nodes from being too close to one another.

$f_2(L) = \sum_{u\in V}((1/{r_u}^2) + (1/{l_u}^2) + (1/{t_u}^2) + (1/{b_u}^2))$ where r_u, l_u, t_u, and b_u are the distances of u from the right, left, top, and bottom borders of the drawing respectively. This is to prevent nodes from positioning too close to the borderlines.

$f_3(L) = \sum_{(u,v)\in E} d_{uv}^2$ This component is to avoid edges from being too long.

$f_4(L) =$ The number of edge crossings in L.

$f_5(L) = \sum_{(u\in V),(e\in E)}(1/g(u,e)^2)$, where $g(u,e)$ is the minimal distance between vertex u and edge e. This penalizes the node and edges that are too close to one another.

The weights on the components of the cost function of Equation 2.8 can be adjusted to obtain drawings with different aesthetics. For instance, a high value of λ_4 results in drawings with few crossings. The computational intensity of the simulated annealing approach to graph drawing makes this algorithm not suitable for interactive applications.

2.2.7 The Time Complexity of the Algorithms

One global iteration of Eades [14], GEM [15], and FR [16] algorithms has a time complexity of $O(n^2)$. The number of global iterations for these algorithms is not theoretically known, but empirically estimated to be of $O(n)$.

For KK algorithm [21], $O(n^3)$ is primarily required to compute the graph theoretic distances between pairs of nodes. The time needed to terminate this algorithm is $O(Tn)$ where T is the total number of iterations of the inner loop and is not theoretically known.

In Tu [36], the cost of computing the permutation of the nodes is $O(|V||E|)$. The cost of computing Tu's cost function when one node changes position is $O(d|E|)$

where d is the number of newly drawn edges. This cost is computed for every node for a number of sample positions proportional to the degree of the node in both initial placement and local optimization phases. Note that in addition, Tu's initial placement involves the recursive optimization of the adjacent nodes of a node whose placement improves its cost.

The second algorithm introduced by Tunkelang [37], is the same as FR in terms of time complexity with the exception of the cost of computing repulsive forces that is $O(n \log n)$ per iteration.

The running-time of DH is noted in [12] as $O(|V|^2|E|)$. This is due to the fact that the algorithm makes $30|V|$ configuration changes each of which is the result of one node replacement with the cost of $O(|V||E|)$.

2.3 Previous Experimental Comparisons

Himsolt et al. [7] performed experiments on GEM, FR, KK, Tu, and DH, and compared their execution time and the quality of their graph layouts. In this Section, we provide a summary of their report. We must note that the graphs used in these experiments have a maximum of 180 nodes and edges in total. In addition, they used a modified version of KK from the Graphlet software as explained in Section 2.2.4.

Running-time

The following is an evaluation presented in [7] on the running-time of the algorithms mentioned above

- GEM and KK are competitive in speed and outperform FR, Tu, and DH.
- FR is fast on small graphs and slows down for large graphs.

Our experiments show that GEM becomes slower than FR as the graphs become larger. See Appendix B for details of our experiment.

Quality

Himsolt et al. [7] used the ratio of the longest edge to the shortest edge as well as the normalized edge lengths deviation for their evaluation of the quality of the graph layouts generated by each algorithm. The following is a summary of this evaluation

- FR, KK, GEM, DH without crossing optimization (i.e. without $f_4(L)$ of Equation 2.8) generate layouts with similar appearances. They generated distorted layouts for loosely connected graphs.
- Tu's layouts are different from the others. Particularly, it does not display symmetries. However, it performs well on grid like graphs such as square or triangular grids.
- High weights for edge crossings and close node and edge distances in DH destroy the symmetries, the edge length uniformity, and the node distribution in the layouts of this algorithm. In general, it is difficult to find the right mix of DH's parameters to obtain good drawings.

One criticism that can be made about the layout quality evaluation presented in [7] is that they do not present the number of edge crossings for different graph layouts

generated by the algorithms compared. This is particularly significant for the layouts of planar graphs.

2.4 Java-based Graph Drawing Tools

2.4.1 Visualizing Graphs with Java: VGJ

VGJ [24] is a graph drawing software available as an applet and an application. It supports three algorithms: the KK Spring algorithm [21], Walker's tree algorithm [40], and CGD algorithm for drawing directed graphs [25]. In addition, it supports graph editing. The applet version of VGJ is not signed[2] [10], and therefore, does not support local file I/O. In addition, although the software provides three algorithms for drawing a variety of graphs, it is not very suitable for experimentation with different drawing algorithms. It does not provide the running-time of an algorithm, does not include modules for evaluating the quality of its graph layouts based on different aesthetic criteria, and does not distinguish between node-based and iteration-based animations (see 'The Remaining Classes' Sub-section of Section 4.4.4) which are very useful for experimentation with force-directed algorithms. In addition, the VGJ applet is not loaded using a JAR, Java Archive file [10], in one Internet transaction. This can slow down the execution of the applet since anytime a class is used for the first time, it has to be loaded from the server side.

[2]Due to Java security restrictions a regular applet cannot access files on the local disk. For this reason, VGJ applet does not allow load and save operations. Although this restriction exists for applets by default, but Java has provided ways for permitting an applet to access a local disk and have all the privileges that a Java application has. An applet with such privileges is a signed applet.

2.4.2 Brown University's Graph Drawing Server: GDS

GDS, graph drawing server [8], is available as an applet and an application for graph editing, graph/drawing format translation, and graph layout generation. GDS provides very complete editing facilities. The software includes a variety of algorithms for drawing general graphs, orthogonal graphs, and directed hierarchical graphs but no Spring algorithm. The graph drawing algorithms of GDS applet are executed on the server machine. This slows down the running-time of these algorithms specially for large graphs. In addition, GDS applet is not signed, and the users cannot perform local file I/O. For saving files, the applet returns the URL of the file that contains the graph layout (produced on the server side) to the user. The GDS applet is not loaded using a JAR file, hence as mentioned in Section 2.4.1, this can slow down the execution of the applet. Furthermore, GDS too does not aim at providing facilities for experimentation with different graph drawing algorithms.

2.4.3 Tom Sawyer Software

Tom Sawyer Software provides the widely known commercial GRAPH LAYOUT TOOLKIT(GLT) and GRAPH EDITOR TOOLKIT(GET) [13]. The company has recently released the Graph Editor Toolkit for Java, version 3.0. Some of the features of this software include

- Automatic drawing of graphs.
- Smart graph labeling.
- Graph editing utilities.

Chapter 3

A Cost-oriented Spring-based Layout Algorithm

As pointed out in Chapter 2, the existing force-directed spring-based graph layout algorithms, such as GEM [15] and FR [16], suffer from inaccurate temperature scheduling and stopping criteria. In this Chapter, we describe our approach which uses node layout and graph layout qualities to determine the local temperature of a node and to detect the convergence of our algorithm respectively. We start by defining these qualities, and explaining the importance of efficient functions for measuring the quality of a node or a graph layout in different stages of a Spring algorithm. Then we compare a number of different possible choices for these functions, and introduce ours which we call the *node layout cost function* and the *graph layout cost function* respectively. Finally we describe our Spring algorithm that is based on these cost functions and discuss the running-time of this algorithm.

3.1 Node and Graph Layout Qualities in Spring Algorithms

In this section, we define node and graph layout qualities, and discuss the importance of measuring the desirability of a node position and a graph layout as a whole. In this discussion, *Spring algorithms* refer to those algorithms of this family that define the negative gradient of an implicit cost function in terms of the forces that are the same or similar to what was defined in Section 2.1.1. GEM and FR are two examples of such algorithms.

3.1.1 What are Node and Graph Layout Qualities?

Each graph drawing algorithm aims at optimizing certain aesthetic criteria. These aesthetics may include: minimization of edge crossings, preservation of graph symmetries, or bend minimization [30]. As a result, the quality of a graph layout is an evaluation of the extent to which a drawing meets certain aesthetic criteria.

As explained in Chapter 2, Spring algorithms move nodes to new positions to minimize the local energy according to their "force laws". These force laws bring adjacent nodes to a distance approximately equal to a predefined optimal edge length, and move non-adjacent nodes to at least a distance equal to this optimal edge length (formal definition can be found Section 2.1.1). Thus, the distances of nodes to their neighbors and their non-neighbors can be used for measuring a node or a graph layout quality. In addition, Spring algorithms explicitly minimize the energy (i.e.,

the magnitude of the overall force acting on each node[1]), and implicitly increase its symmetries and decrease its number of edge crossings. Therefore, other aesthetic criteria for measuring the graph layout quality for this family of layout algorithms are the overall energy of the graph layout, the preservation of symmetries in the drawing, and its number of edge crossings.

Furthermore, in addition to measuring the quality of a graph layout as a whole, we can also evaluate the layout quality of each node individually. Some aesthetic criteria such as closeness of neighbors and non-neighbors, or the energy of nodes allow for evaluation of individual node layout qualities. As a matter of fact, in the case of these two criteria the overall quality of the graph may be defined as a function of the individual node layout qualities. For instance, the overall energy of the graph can be defined as the average or the maximum energy of the individual nodes. In the rest of this Section, we explain where the node and graph layout qualities can be used in Spring algorithms, and discuss the general form of measuring them.

3.1.2 Applications of Measuring a Node or a Graph Layout Quality

In the execution of a Spring algorithm, a good criterion function to measure the quality of a node location can be used for:

[1]Kamada and Kawai use a different definition of energy in their algorithm. For this, please refer to Section 2.2.4.

- Stopping the local movements of a node that moves more than once per global iteration[2].
- Stopping the global iterations of the algorithm.
- Choosing a node temperature.

In addition, measuring the quality of a graph layout allows for automatic selection (i.e., without any human interference) of the layout with the best quality among a number of different layouts for the same graph. This is especially useful for the graph layouts produced by a Spring algorithm. This is because, depending on the initial layout of the Spring algorithm, the resulting layout may end up in some unwanted local minimum. Thus, we can use the graph layout qualities to compare early or intermediate results of executing Spring algorithm on a number of different initial layouts of the same graph, and continue the Spring algorithm only on the best layout selected. Although, such an approach may increase the running-time of finding a graph layout, it can reduce the chance of arriving at an undesirable local minimum.

We emphasize that the methods which we are seeking for evaluating a single node quality or the entire drawing quality should be efficient. This is due to the fact that we are improving on the algorithms such as GEM or FR that already have a running-time of $O(n^2)$ per iteration.

[2]In some Spring algorithms, a node is repositioned more than once per global iteration. The forces on the node are computed and the node is moved until its location stabilizes according to a stopping criteria.

3.1.3 Cost Function for Measuring Node and Graph Layout Qualities

We refer to *cost* as a value that represents the quality of a node or a graph layout. A *cost function* computes this cost value. A cost function that evaluates the quality value of a node v from a layout L of graph G, has the following general form

$$Q_v = nodeLayoutCost(G, X, Y, v) \tag{3.1}$$

and the corresponding function for quality of layout L takes the form of

$$Q_L = graphLayoutCost(G, X, Y) \tag{3.2}$$

where G holds the adjacency information of the graph, X and Y are respectively the vectors of x and y coordinates of graph vertices in layout L. Q_v and Q_L are numbers that represent the quality values of the node layout of vertex v and of the graph layout L respectively. As mentioned in Section 3.1.1 $graphLayoutCost$ may be defined as a function of node layout qualities.

For a given graph, the values of Q_v for each node and Q_L for a graph layout generated by Spring algorithm are not known prior to the execution of a Spring algorithm. Our approach is to use the differences of Q_v and Q_L values of Equations 3.1 and 3.2 obtained from successive iterations (in other words, detect their convergence) to use in the applications of measuring node and graph layout quality mentioned in Section 3.1.2. We elaborate on this in Section 3.3.

Different cost functions in the form of the Equations 3.1 and 3.2 have been defined by Davidson and Harel [12] and Kamada and Kawai [21]. We evaluate these further

in the following Section.

3.2 Choice of the Cost Function

In this Section, we are evaluating a number of different choices for our *cost functions*. Our goal is to find the functions that fulfill the requirements mentioned in Section 3.1.2. The advantages and disadvantages of each choice are discussed, and the functions which we find to be most suited for our purpose are introduced at the end of this Section.

We use the terms: Q_v, Q_L introduced in Section 3.1.3 for each choice that we are evaluating. In each choice, we present a function definition for these terms.

3.2.1 Magnitude of the Overall Force as Cost

In this Section, we look at the possibility of representing the quality value of each vertex of a layout L of graph G with the magnitude of the overall attractive and repulsive forces acting on that vertex. The quality value of the layout L as a whole will be a function of the quality values of the graph vertices. The overall forces on vertex v is defined as

$$F(v) = \sum_{(u,v)\in E} \frac{d_{uv}^2}{k}(\frac{u-v}{d_{uv}}) - \sum_{(u,v)\in V^2} \frac{k^2}{d_{uv}}(\frac{u-v}{d_{uv}}) \tag{3.3}$$

where d_{uv} is the distance between u and v, and k is the optimal edge length.

Thus, the quality of vertex v is presented as

$$Q_v = |F(v)| \tag{3.4}$$

and Q_L, the quality of a layout L with vertices $v_1, ..., v_n$ is defined as

$$Q_L = f(|F(v_1)|, |F(v_2)|, ..., |F(v_n)|) \tag{3.5}$$

where f is a function such as the mean or the maximum of the force magnitudes.

Advantages

Using the magnitude of the overall force acting on a vertex v seems to be a natural choice for measuring the quality of v. This is because this force is computed for each node in each iteration of a force-directed Spring algorithm; and therefore, no extra computation is required. Starting from the initial random graph layout L, Q_v for vertex v of this layout is initially large and as the attractive and repulsive forces on v become more balanced this value decreases. Figure 3.1 shows the values of Q_L where this function computes the maximum of overall force magnitudes of vertices in five different graphs over 500 iterations of an implementation of the FR Spring algorithm[3] [4]. Detecting the convergence of the overall force magnitude for vertex v can be used in the applications of the node quality measurement mentioned in Section 3.1.2. However, as we will see in the rest of this Section using magnitudes of forces as a measurement for quality has a number of major drawbacks.

[3]The graph layouts have been generated in Matlab [1] using the Graphlet [20] implementation of FR Spring algorithm translated from C++ to Matlab script. The default configurations of Graphlet version of FR are used except for the number of iterations that is fixed at 500.

[4]The binary tree with 31 nodes is in an unpleasing local minimum, but as we can see the maximum force magnitudes of its nodes converges to a small value near 3.

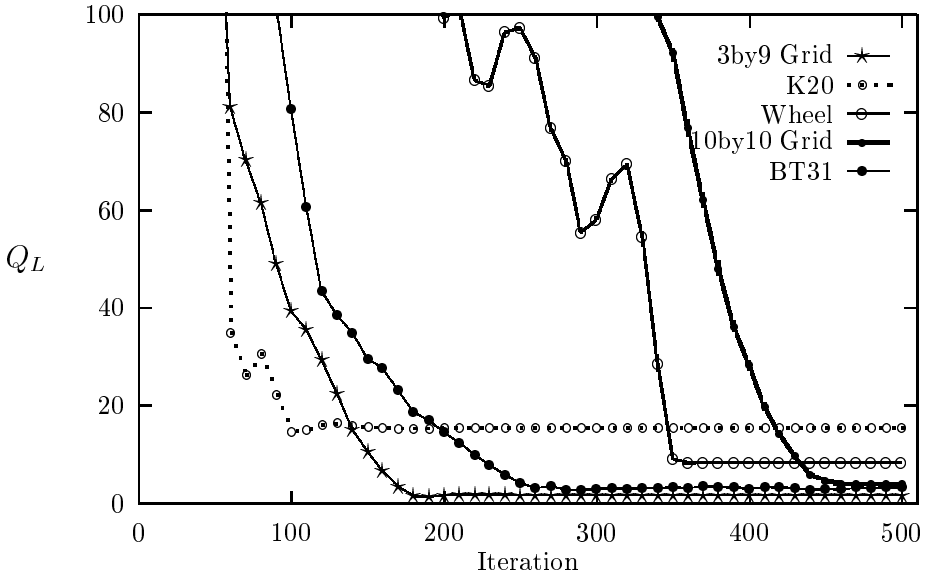

Figure 3.1: Q_L at every 10 iterations of FR. Q_L is the maximum of overall force magnitudes.

Disadvantages

One observation that we have made about the force magnitudes on the vertices of a graph layout is that they converge late during a Spring algorithm. In other words, a graph layout may not improve further significantly, but the force magnitudes on its vertices still decrease by large amounts. This is a drawback for using Q_L introduced in this Section for stopping a Spring algorithm.

Another disadvantage of using the cost function of Equation 3.5 for measuring the quality of a graph layout is that each argument to this function is dependent on the scaling of its layout. The Q_v value for each vertex v is dependent on the scaling of L. This means that if layout L_2 of graph G is the scaled version of layout L_1 of the same graph then the overall force magnitudes on corresponding vertices of L_1 and L_2 may

be different. If we scale the layout L_1 by a factor of α and obtain a second layout L_2, the overall force on vertex v of L_2 will be the following

$$F(v) = \alpha^2 * \sum_{(u,v)\in E} \frac{d_{uv}^2}{k}(\frac{u-v}{d_{uv}}) - \frac{1}{\alpha} * \sum_{(u,v)\in V^2} \frac{k^2}{d_{uv}}(\frac{u-v}{d_{uv}}) \tag{3.6}$$

where d_{uv} is the distance between u and v in L_1.

This drawback can be illustrated by the layouts in Figure 3.2. It is obvious that layout (a) is better than layout (b) in Figure 3.2. However, this is a drawback, since the Q_L value of a scaled version of graph layout (a) of Figure 3.2 may be larger than the Q_L value of graph layout (b) of the same Figure. In this case, another graph layout quality criterion such as the number of edge crossings should be used to automatically choose the better graph layout among the two layouts.

The other disadvantage of the cost function of Equation 3.5 is its inability to detect that some nodes of a graph are in an undesirable local minimum in a graph layout. Figure 3.2 presents the value of Q_v for each vertex v of two graph layouts of a 3×9 five-point operator grid (note that the forces are rounded off to their nearest integer values). The graph layout on the left is a crossing-free drawing of the grid, and the one on the right is a drawing of the same graph that is in an undesirable local minimum. The Figure shows that there is a small difference between the magnitude of the forces in the two drawings. Table 3.1 presents the unrounded minimum, maximum, and mean of the force magnitudes of the vertices in the two graph layouts. Clearly, the differences are negligible. Indeed, the graph layout with undesirable local minimum has lower minimum and maximum force magnitudes.

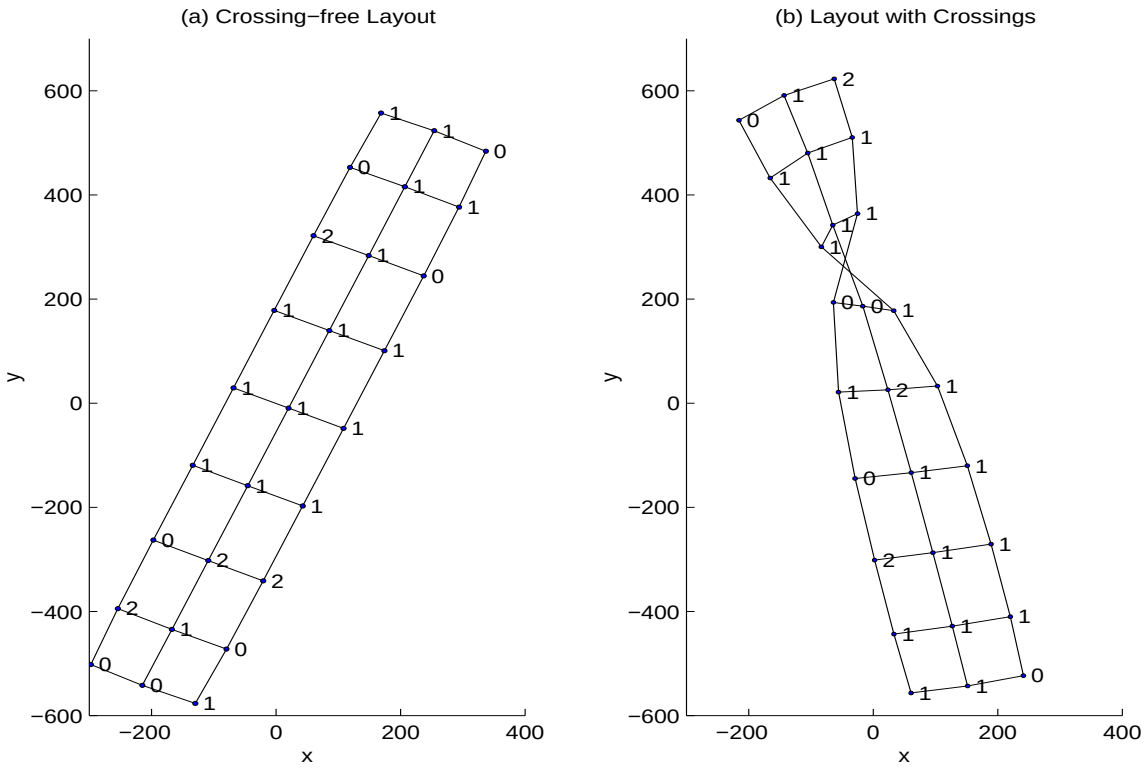

Figure 3.2: Two Graph Layouts of a 3×9 Grid With the Magnitude of the Overall Forces on Vertices Displayed.

This phenomena is due to the fact that the optimization process of force-directed Spring algorithms always seeks for a low energy drawing (i.e., minimum overall force magnitudes on vertices). However, in many cases, it is required to move through a worse drawing (i.e., with larger forces on vertices) before finding a better drawing [16]. This process is called *hill climbing* in simulated annealing[22, 28]. When a Spring algorithm finds an undesirable local minimum in which the forces are small, such a drawing still needs to go through a hill climbing process. However, if the forces on vertices are small then the vertices won't have the ability to make the large enough movements to come out of the unwanted local minimum. The second graph layout of

Force Magnitudes	*Layout (a)*	*Layout (b)*
Maximum	1.8721	1.8351
Minimum	0.1203	0.0940
Mean	0.9038	0.9182

Table 3.1: The maximum, minimum, and average force magnitude of graph layouts of figure 3.2.

Figure 3.2 is a result of this.

This is a drawback for using the cost function of Equation 3.5 for automatically choosing the best layout among a number of layouts of the same graph. This is one of the applications of measuring graph layout quality mentioned in Section 3.1.2.

3.2.2 Kamada and Kawai's Energy as Cost

Another choice for measuring the layout qualities of a node and a graph is by using Kamada and Kawai's [21] energy function. They consider all the nodes in the graph as particles where every pair of particles is connected by a spring. They use actual and graph-theoretic distances between pairs of nodes to re-position the nodes so that the spring system is balanced (see Section 2.2.4). The energy of their system, E, is defined by Equation 2.7. Here we are not using this Equation for finding the positions of vertices which was the goal of KK algorithm. Instead, we intend to use Kamada and Kawai's energy formula for defining the cost functions that we are seeking to measure node and graph layout qualities.

We consider E of Equation 2.7 to indicate Q_L, the quality of layout L of graph G

$$Q_L = E \tag{3.7}$$

and based on Equation 2.7, we define, Q_v the quality of vertex v, as following

$$Q_v = \sum_{u \neq v} \frac{1}{2} k_{uv}(d_{uv} - l_{uv})^2 \tag{3.8}$$

Advantages

The advantage of using Q_v of Equation 3.8 for measuring the quality of a node v is similar to what was mentioned about using the magnitude of the forces in Section 3.2.1. Over iterations of a Spring algorithm as the distances between pairs of nodes stabilize, Q_v value for each vertex v will converge. This makes this function suitable for the applications of the cost function for the node quality measurement mention in Section 3.1.2.

The preprocessing phase for finding l_{uv} and k_{uv}, and computing the sums in Equations 2.7 and 3.8 do not change the time complexity of one iteration of a Spring algorithm. However, as we will see in what follows, these computations add a substantial cost to the running-time of a Spring algorithm.

Disadvantages

The first disadvantage of using Q_L as defined in this Section, as in the case where using the magnitude of overall force as cost, is that this Equation is dependent on the scaling of a graph layout L. The drawback of this is explained in Section 3.2.1.

The second disadvantage of using the functions of Equations 2.7 and 3.8 for measuring the qualities of a node and a graph layout has to do with the amount of computation required. Computing these new Q_L and Q_v substantially adds to the

running-time of a Spring algorithm which has been empirically known to be $O(|V|^3)$ (see Section 2.2). We elaborate on this in the following.

We start by the time complexity of the preprocessing stage to compute l_{uv} and k_{uv} for each pair of nodes in the graph G. l_{uv} is the length of a shortest path between u and v in G multiplied by a constant value. k_{uv} is the strength of the spring between u and v which is the reverse square of the length of the shortest path between u and v in G multiplied by a constant value. For a graph with n nodes, the time complexity of finding all shortest paths is primarily $O(n^3)$ [11]. For large graphs it is possible to reduce this to $O(n^2 \log n)$, or $O(n^2 \log^2 n)$ [26, 31]. Therefore, we can see that the additional cost of computing l_{uv} and k_{uv} using the most efficient implementation is $O(n^2 \log n)$. In addition, we add the arithmetic expression: $\frac{1}{2}k_{uv}(d_{uv} - l_{uv})^2$ for n^2 times to one iteration of a Spring algorithm (i.e., n times for each node). The only saving that we can make in this case is by reusing d_{uv} that is originally computed for obtaining the attractive and repulsive forces between u and v. Thus, using the energy function of Kamada and Kawai for measuring node and graph layout qualities adds a substantial amount of computation to a Spring algorithm, even though it does not change the time complexity of this algorithm.

3.2.3 Davidson and Harel's Cost Function

For completeness, we would also like to briefly mention the cost or target function of the simulated annealing approach of Davidson and Harel to graph drawing [12]. As summarized in [7], the following presents the general form of their cost function for a

layout L of graph G

$$f(L) = \sum \lambda_i f_i(L) \tag{3.9}$$

where the f_i elements of the sum, along with their time complexities, are[5]:

1. Node Distribution: $O(|V|^2)$.
2. Borderlines: $O(|V|)$.
3. Edge Lengths: $O(|E|)$ since edge lengths computed for node distribution can be reused.
4. Edge Crossings: $O(|E|^2)$.
5. Node-Edge Distances: $O(|V||E|)$.

and λ_i is the weight of each of the above components, and it can be set by the user.

An advantage of the cost function of Equation 3.9 is the variety of graph layout elements that it takes into consideration for measuring quality. Specifically, the function includes five different quality criteria. However, as we see from the running-times above, this advantage comes with a high expense. The total running-time of finding the cost of graph layout L using the cost function of DH is: $O(|V|^2 + |V||E| + |E|^2 + |V| + |E|)$. The high cost of computing this function makes it an unattractive choice for *graphLayoutCost* function that we are seeking. In addition, the cost function in Equation 3.9 measures a graph layout quality and not a node layout quality. The

[5]The time complexities are based on the actual definitions of the components defined by Davidson and Harel in [12] (see Section 2.2.6).

only components of the function that can be computed for an individual node are: the borderlines and the node-edge distances. Borderlines are not very important and are eliminated in some implementations of this cost function [7]. Furthermore, the computation of node-edge distances are too costly to be considered for *nodeLayoutCost* function.

3.2.4 Our Cost Function

In this Section, we introduce our node and graph layout cost functions and discuss some of their properties. Our goal is to find such functions that can efficiently measure a node layout cost over the iterations of a Spring algorithm, and a graph layout cost to evaluate the quality of a number of layouts of a given graph to fulfill the applications mentioned in Section 3.1.2.

The general forces used in a Spring algorithm such as FR [16] or GEM [15] aim at bringing the adjacent vertices of a node to a close and uniform distance to the node (approximately equal to a predefined optimal distance), and move the non-adjacent nodes away from the node. This intuition on the Spring algorithm has been the motivation behind our function that computes Q_v and is defined below. Furthermore, we define Q_L based on the Q_v values of all the vertices in a graph.

Terminologies

In terms of graph:

- $Adj(v)$: the set of nodes adjacent to v
- $Adj'(v)$: the set of nodes non-adjacent to v

In terms of graph layout:

- $edist_L(v,u)$ = Euclidean distance between v and u in the graph layout L

  or

  $edist(v,u)$ if the graph layout L is clear from context

- $edist(v,U) = min(edist(v,u) : u \in U)$

- $EDIST(v,U) = max(edist(v,u) : u \in U)$

Definition

For each vertex v in a layout L of graph G, we define Q_v as

$$Q_v = \frac{EDIST(v, Adj(v))}{\alpha * edist(v, Adj(v)) + (1-\alpha) * edist(v, Adj'(v))} \tag{3.10}$$

where $0 < \alpha < 1$. In this thesis, we chose α to be 0.5 since this will give an equal weight to the denominator components of Equation 3.10.

If v is adjacent to all vertices in the graph (i.e., $Adj'(v)$ = empty) it is reasonable to define

$$edist(v, Adj'(v)) = \text{width of layout frame} \tag{3.11}$$

or

$$edist(v, Adj'(v)) = EDIST(v, Adj(v)) \tag{3.12}$$

The choice of Equation 3.12 is used in our implementation of CostSpring algorithm. For a given graph layout L, we obtain its Q_L value as

$$Q_L = max\{Q_v : v \in G\} \tag{3.13}$$

Properties

With regards to the components of Equation 3.10, we can also think that around each vertex v, there exist three balls centered at v

- The first with radius $EDIST(v, Adj(v))$ which is the smallest ball that will enclose all adjacent vertices of v.
- The second with radius $edist(v, Adj'(v))$ which is the largest ball that will exclude all non-adjacent vertices of v.
- The third with radius $edist(v, Adj(v))$, which is the largest ball that will exclude all adjacent vertices of v.

Spring algorithms aim at finding graph layouts in which for each vertex v

$$edist(v, Adj(v)) \simeq EDIST(v, Adj(v))$$

and

$$EDIST(v, Adj(v)) \leq edist(v, Adj'(v)).$$

Therefore for a layout L of graph G generated by a Spring algorithm, this results in small values for Q_L and Q_v for all $v \in G$. Appendix A shows five graph layouts produced by the FR Spring algorithm. For each layout, maximum Q_v (i.e., Q_L) and minimum Q_v are recorded.

Over the iterations of a Spring algorithm as the distances between a node and its neighbors and non-neighbors stabilize, the Q_v of each vertex v converges. Furthermore, if the scaling of all the distances to adjacent and non-adjacent nodes to a

vertex changes by the same factor, the Q_v value for that node remains the same. The convergence of *nodeLayoutCost* function introduced in this Section over the layouts generated during the execution of a Spring algorithm allows this function to be used in the applications mentioned in Section 3.1.2 for node quality measurement. Figure 3.3 shows the Q_L values of Equation 3.13 for the same graphs with the same initial layouts used in Figure 3.1[6]. The algorithm, environment, and configurations used are the same as the ones applied to obtain the data for Figure 3.1.

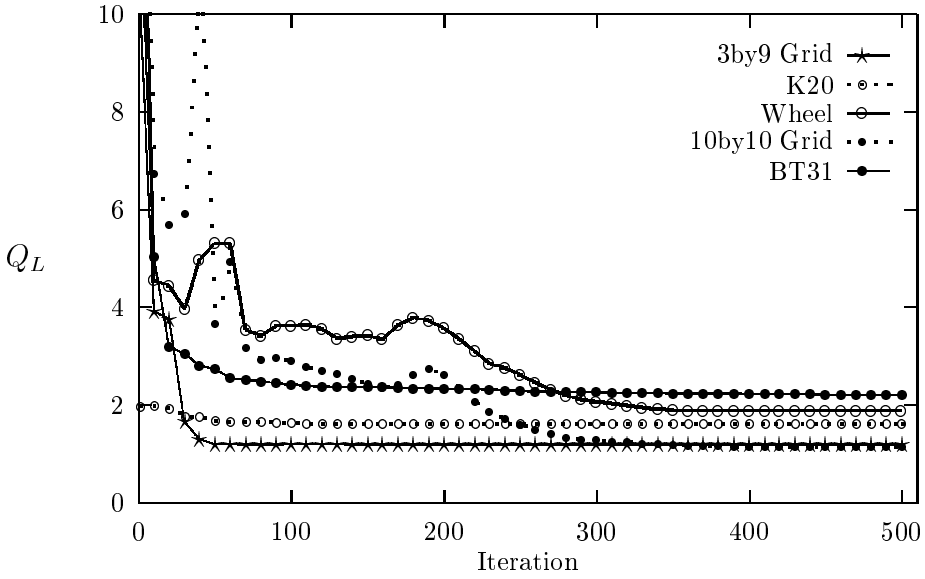

Figure 3.3: Q_L of Equation 3.13 at every 10 iterations of FR.

[6]The binary tree with 31 nodes is in an unpleasing local minimum, and that is why Q_L converges to a value greater than 2. A pleasing drawing of the same graph produced by the same algorithm converges to a value near 1.

Advantages

Convergence: Figures 3.1 and 3.3 show that Q_L that is introduced in this Section converges earlier over the iterations of a Spring algorithm than the one introduced in Section 3.2.1. We have also observed that the quality of the graph layouts do not improve significantly once Q_L converges. This is an advantage for using Q_L for stopping a Spring algorithm.

Insensitivity to scaling: Q_v for all $v \in G$ and as a result Q_L are not dependent on the scaling of a layout of graph G, since Q_v of Equation 3.10 is a ratio of one distance over the sum of two other distances in a layout of graph G. Dependability on scaling was the drawback of the Kamada and Kawai's cost function and the magnitude of forces as the cost function. As mentioned in Section 2.2.3, the GEM [15] algorithm explicitly tries to detect nodes that are rotating and brings their temperatures down to indirectly stop the algorithm. The Q_v values for each node v of rotating graph layouts are also the same. Therefore, if the algorithm is rotating a layout, we will detect convergence using our cost function.

Detection of an unwanted local minimum: Our experiment has shown that the Q_L of Equation 3.13 is useful in distinguishing which layout is in an undesirable local minimum among two or more layouts of the same graph (see Section 5.6). Figure 3.4 shows the two graph layouts of Figure 3.2 with the rounded Q_v values for all vertices in the two graph layouts. As we can see, unlike the magnitude of forces, our Q_v values for the nodes surrounding the twisted area of graph

Our nodeLayoutCost	*Layout (a)*	*Layout (b)*
Maximum	1.1938	2.6526
Minimum	0.9627	0.9583
Mean	1.1179	1.4398

Table 3.2: The maximum, minimum, and average Q_v values of Equation 3.10 for graph layouts of Figure 3.4.

layout (b) are higher than the rest of the nodes in the layout. Table 3.2 shows the maximum, minimum, and the average Q_v values of the vertices in the two graph layouts of Figure 3.4.

Fruchterman and Reingold [16] note that when a graph layout has an undesirable local minimum some nodes become a barrier against the movements of some other nodes which need to pass over this barrier to achieve a better layout. Therefore, there are often adjacent nodes located on the opposite sides of this barrier. We observe that this usually causes the maximum distance between the adjacent nodes located on the two sides of the barrier to become large, and the minimum distance to a non-adjacent node for the vertices in the barrier as well as the vertices trying to pass over the barrier becomes small. These can increase the Q_v values of the nodes involved in the undesirable local minimum.

Efficiency: Given a graph layout L the cost of computing Q_v for all nodes is $O(|V|^2)$. This includes the cost of

- Identifying the neighbors and non-neighbors for all node: $O(|V|^2)$.

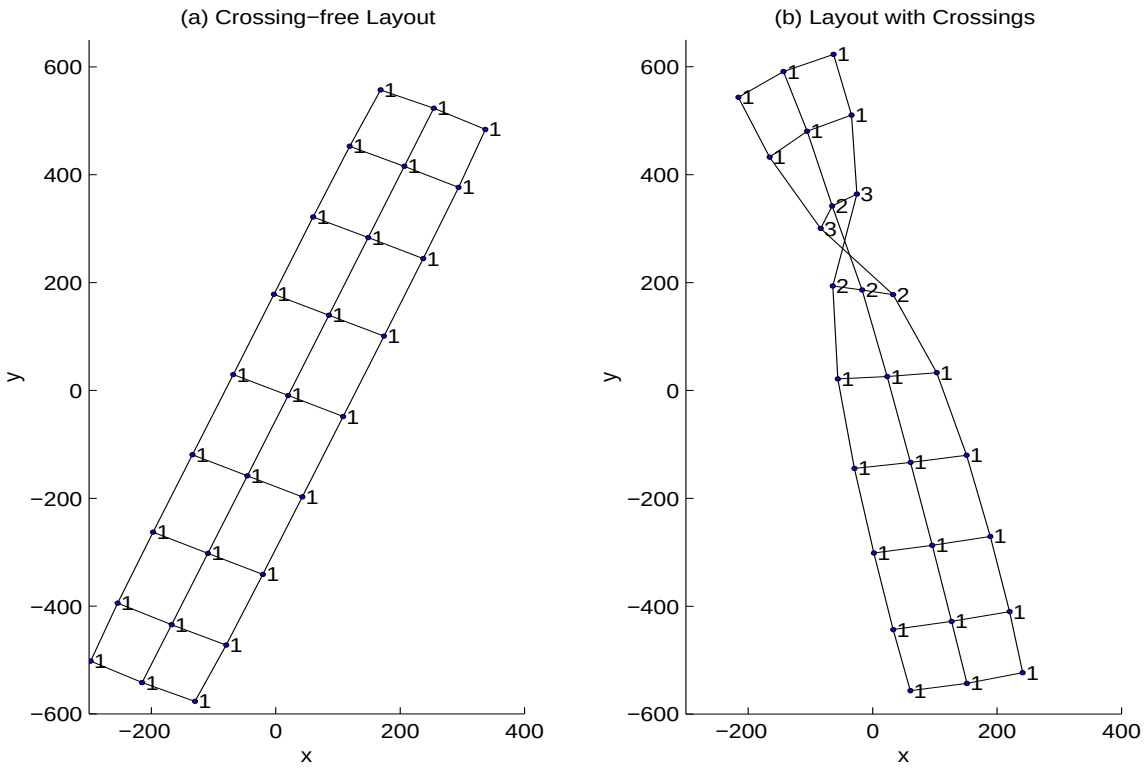

Figure 3.4: Two Layouts of a 3×9 Grid With the Q_v of Equation 3.10 displayed for each node.

- Finding $EDIST(v, Adj(v))$, $edist(v, Adj(v))$, and $edist(v, Adj'(v))$ for each vertex v: $O(|V|^2)$.
- Finally, computing the Q_v value for each vertex v: $O(|V|)$.

The first item mentioned above is computed only once in a preprocessing stage. As we would need the Q_v value of each node in every iteration of a Spring algorithm, the computation of the second and the third items are repeated for every iteration of the algorithm. The cost of one iteration of a Spring algorithm is $O(|V|^2)$. This makes the second item a potentially costly operation. However, we reuse the distances between pairs of nodes that are computed during the

computation of forces on each node for the second item above. This only adds a maximum of two comparisons and one assignment where the attractive and repulsive forces from an adjacent node are computed, and adds a maximum of one comparison and one assignment where the repulsive force from a non-adjacent node is computed. This is a negligible amount of computation compared to the computation of attractive and repulsive force, and thus it is computationally acceptable.

After each iteration for stopping the algorithm, we need to compare the Q_v values of the nodes in graph layout L produced by this iteration with the corresponding values from the previous iteration to detect the convergence of the Q_v values of the graph vertices. A Q_v value computed during an iteration in the fashion explained above is an approximation to the actual Q_v of vertex v in the graph layout L in hand after that iteration. We elaborate on this further in Section 3.2.5.

3.2.5 An Approximate Cost Value

In previous subsections of this Section, we suggested to reuse the distances computed during force computations of a Spring algorithm in finding the respective Q_v values of the vertices in the graph layout at hand. This provides an accurate value of Q_v for vertex v at the time when Q_v is computed. However, we also suggested to use the Q_v values computed in this fashion during an iteration, as Q_v values of vertices in the final graph layout L produced by this iteration. In Spring algorithms such as GEM [15], where the nodes are moved one at a time, this Q_v is an approximation to

the actual Q_v in graph layout L. This is because, depending on where v falls in the order that the new positions of nodes are computed, there are i nodes, $1 \leq i \leq |V|$, that are moved after v (this includes v itself).

In Spring algorithms such as FR [16] the nodes are repositioned all at once, and precisely after the computation of forces for all nodes are completed. Therefore, the effect of using Q_v values obtained using the distances found during the computation of the forces is that: the Q_v of vertex v after iteration x is really the accurate Q_v after iteration $x - 1$.

In our cost-oriented Spring algorithm described in the rest of this Chapter, we move the nodes one at a time and use an approximate Q_v. As reported in Section 5.3.3, this approximation still provides us with an acceptable result for stopping a Spring algorithm.

3.3 The CostSpring Algorithm for Drawing Undirected Graphs

In this Section, we introduce our cost-based Spring algorithm which we call *CostSpring*. In this algorithm we make use of the node qualities for choosing the node temperatures, stopping the movements of a node that can be repositioned more than once per iteration, and in stopping the global iterations of the algorithm. We recommend this algorithm for graphs with up to 256 nodes, due to the computational intensity of Spring algorithms with time complexity of $O(|V|^2)$ per global iteration. Figure 3.5 is the outline of the two phases of this algorithm. The segment of the al-

gorithm shown in this Figure is repeated once for each phase. Parts of the code that differ in the two phases are highlighted using square boxes and are labeled accordingly for each phase. The functions and important constants used in this segment are surrounded by ellipses and rectangles respectively. The default values of the configurable parameters used in this algorithm can be found in Appendix D.

The input to this algorithm is the adjacency information of the graph that is to be drawn, and the output is the positions of the graph vertices. The attractive and repulsive forces used in this algorithm are those of the FR [16] algorithm. However, unlike their algorithm in each iteration, we compute the forces on nodes and reposition them one at a time. The cost function that we use for each node's quality is the one of Equation 3.10. To compute Q_v for each node, we use the distances between pairs of nodes found during the computation of attractive and repulsive forces for each node. We use a random order of nodes for processing the nodes in the for-loop of Figure 3.5.

Initial Layout

Our algorithm takes an input parameter that indicates whether the positions of vertices are initialized or not. If the positions are not initialized, the algorithm initializes the positions to random locations. Otherwise, the algorithm uses the pre-assigned positions of the nodes. Our algorithm has shown that it can produce good quality of graph layouts from initial random layouts (see Section 5.4). However, there are certain graphs such as grids of large aspect ratio or large sparse graphs such as binary trees and circular graphs that have a higher tendency of ending up in an unpleasing

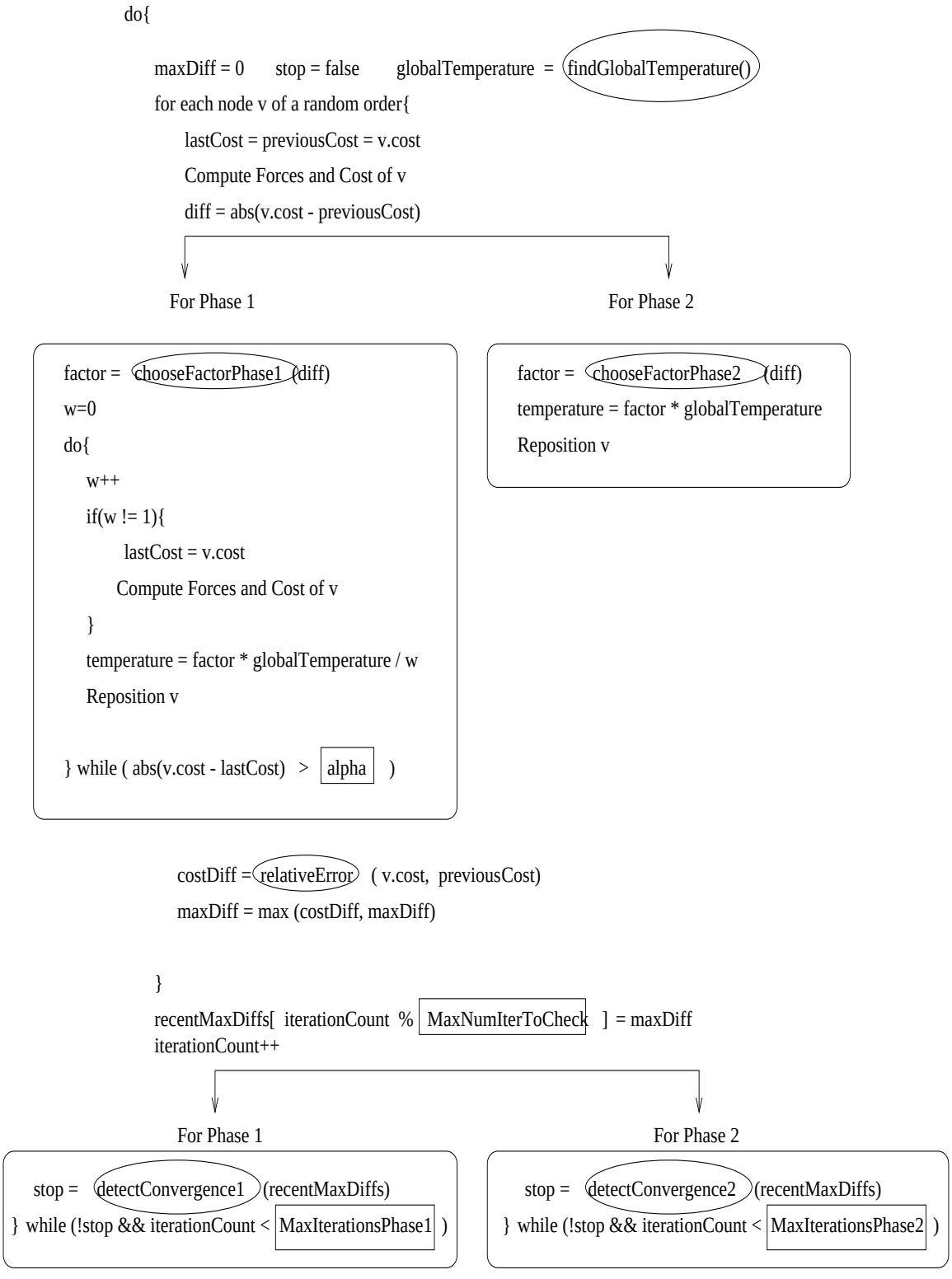

Figure 3.5: Outline of CostSpring algorithm.

local minimum. For these cases, we introduce a new method for finding the initial graph layout of a Spring algorithm using Q_L values.

In this method, we execute a part of the first phase of our Spring algorithm on more than one random initial layout of the graph that is to be drawn. Then we use Q_L plus a penalty for each edge crossing from each resulting layout to find the best graph layout, and continue the algorithm only on that layout. This approach to finding the initial graph layout is costly[7], however, it can decrease the chance of finding an undesirable local minimum (see Section 5.6). The number of initial graph layouts and iterations used to obtain the intermediate graph layouts is crucial to the cost of this approach as well as to the chance of eliminating the unpleasing layouts. The higher these numbers are, the higher is the chance of finding the graph layout that is not in an undesirable local minimum, but also the higher the running-time of finding this layout. In Section 5.6, we conduct experiments with different number of initial graph layouts and iterations for a number of different graphs.

The Two Phases

Our CostSpring algorithm has two phases. The first phase allows for large movements and for nodes to move more than once in each iteration if needed. The second phase is for fine-tuning the resulting graph layout of the first phase. In this phase smaller movements are permitted. This approach was inspired by our observation on the behavior of FR and GEM algorithms. Starting from a random layout for a graph, large movements were needed at the beginning for finding the general shape of the

[7] If we choose N initial random layouts and M iteration of our Spring algorithm then the cost of this method is $N \times M$ iterations.

graph layout, and after that only small movements would be sufficient to fine-tune the final drawing. Fruchterman and Reingold [16] also acknowledged this and used the terms 'quenching' and 'simmering' for the two phases respectively. Even though the FR algorithm does not follow the two-phase approach, they reported that as a result of one of their experiments they believed that such an approach would result in better graph layouts in fewer iterations. Our temperature scheduling is similar but slightly different from what Fruchterman and Reingold suggested for the two phases. In the following subsection we elaborate on this.

Furthermore, in phase 1 we allow a node to repeatedly move until its cost difference from the previous position and the current position fall below a small value *alpha*. The motivation for doing this was due to our observation that, starting from initial random layouts, vertices have very large energies in the early iterations of the algorithm. This resulted in, for instance, large repulsive forces pushing a node far away from the rest of its adjacent vertices. With the node temperatures allowing these nodes to take large steps, a node would move to a far and isolated position from the rest of the nodes. We observed that if in these early iterations we allow for such a node to reposition more than once, it would end up in a more balanced position with respect to other nodes, particularly its neighbors. The experiment of Section 5.3.2 shows the average number of iterations of the inner loop for a node in one global iteration of the first phase of CostSpring algorithm.

Temperature

Fruchterman and Reingold suggested a fast cooling from high to low for phase 1 and a constant low temperature for phase 2. Their temperature is uniform or global for all nodes in an iteration. We introduce a combination of a global and local temperature for each node that is to be moved. As we can see in Figure 3.5, a node temperature is a function of the variables: $globalTemperature$, $factor$, and w. The function $findGlobalTemperature$ computes the global temperature. We choose the global temperature to be a function of the maximum of the overall force magnitudes on vertices after each iteration. Specifically, $findGlobalTemperature$'s body takes the following form

```
findGlobalTemerature(){
   return sqrt(maxForce) / C1
}
```

where $maxForce$ is the maximum force magnitude on the vertices of the graph. *C1* is a constant value that is one of the adjustable parameters of CostSpring algorithm. The parameter $factor$ that is determined by $chooseFactorPhase1$ or $chooseFactor-Phase2$ would possibly raise the $globalTemperature$ by a factor that is determined by the difference in current cost of a node and its previous cost value. The larger this difference is, the larger is the factor that is selected. A factor with value of one is possible. In addition, the range of the factors to be selected in the first phase are larger than the ones of the second phase. The following presents the body of these two functions

```
chooseFactorPhase1(diff){
   if(diff > beta)
     return C2
   else
     return C3
}
```

```
chooseFactorPhase2(diff){
 if(diff > gamma)
     return C4
   else
     return C5
}
```

where $beta$, $C2$, $C3$, $gamma$, $C4$, and $C5$ are constants and can be adjusted under different configurations of the algorithm. The three constant parameters of $chooseFactorPhase1$ are larger than the corresponding ones from $chooseFactorPhase2$. Finally, w is used to gradually decrease the temperature of a node that is moving more than once per iteration, and it is only used in phase 1.

Stopping

To stop the outer do-loop of each phase, we find the maximum of relative errors of all nodes' previous costs to their current costs (see Figure 3.5). Then at the end of the iteration, the function $detectConvergence$ takes these maximum relative error values for the last $MaxNumIterToCheck$, and simply checks if all their values fall below a tolerance value. If they do, this function returns true, in which case the loop terminates. Otherwise it will return false and the loop continues. In cases where our method of terminating a loop fails, we provide the upper bounds $MaxIterationPhase1$ and $MaxIterationPhase2$ to prevent infinite loops.

Originally, we found the relative error of the maximum node cost in the previous iteration to the maximum node cost in the current iteration (i.e., the relative error of

Q_L values) as the $maxDiff$. However, we find our current approach is more accurate for stopping the loops. This is because, currently, we find the largest relative difference among the differences in current node costs and the ones from the previous iteration. In our first approach, it was possible that the difference between the maximum node cost of one iteration and its previous one would not represent the maximum change among the cost values of nodes in these two iterations. The amount of this change indicates how much the layouts of a graph are still improving by the algorithm.

The tolerance values, $tol1$ and $tol2$ for the two phases respectively, that $maxDiff$ values are compared against will affect the number of iterations that each phase would execute. The smaller these tolerance values are, the larger are the number of iterations of each phase. $tol1$ and $tol2$ are respectively used by $detectConvergence1$ and $detectConvergence2$ functions. Generally, $tol1$ is higher than $tol2$. Our experiments also indicate that the graphs with different densities require different number of iterations spent in each phase. Sparse graphs need more iterations of the first phase and very few iterations of the second phase, and dense graphs require few iterations of the first phase and more iterations of the second phase. Thus, our algorithm in a preprocessing stage detects the density of a graph, and chooses the values of $tol1$ and $tol2$ based on that density. The following is an outline of what we consider sparse, intermediate, and dense graph

$$\rho = density = \frac{|E|}{|V|} \tag{3.14}$$

- **Sparse:** $\rho \leq 1.1$**,**

- **Intermediate:** $1.1 < \rho \leq 3$**,**
- **Dense:** $3 < \rho$**.**

3.4 The Running-time of the Algorithm

The following are the running-times of various parts of the algorithm of Figure 3.5 where $n = |V|$; and we will use these running-times to discuss the overall running-time of this algorithm

- findGlobalTemprature(): $O(n)$.
- Compute forces and cost of v: $O(n)$.
- chooseFactorPhase1: $O(1)$. $chooseFactorPhase2$ has the same time complexity.
- relativeError(): $O(1)$.
- detectConvergence1(): $O(MaxNumIterToCheck)$. $detectConvergence2()$ has the same time complexity. $MaxNumIterToCheck$ is normally a small number that in our case is 10.

The running-times of the outer do-loops of phase 1 and 2, and the inner do-loop of phase 1 are not theoretically known. However, our experiment in Sections 5.3.2 shows that the average number of iterations of the inner do-loop of phase 1, for each node and for the graphs used in that experiment, is 1.3627. Therefore, based on this experiment

and the time complexity of the components stated above, the running-time of the for-loop of this algorithm is $O(n^2)$ for both phases. The number of iterations that the outer loop can execute in the worst case is determined by $MaxIterationsPhase1$ and $MaxIterationsPhase2$. In our implementation, the total value of these two variables is $5n$. Therefore, the worst case running-time of this algorithm is $O(n^3)$. The value $5n$ here is the maximum number of iterations for the outer loop which normally is not reached. The experiment of Section 5.3.4 shows that the average number of iterations of the outer do-loop is $2.17n$.

Chapter 4

Design and Implementation of LayoutShow

4.1 LayoutShow: a Signed-applet/Application for Graph Drawing and Experimentation

LayoutShow is a Java-based multi-threaded applet/application for experimentation with different graph drawing algorithms [5]. It supports a variety of spring-based graph drawing algorithms as well as layouts based on eigenvectors. The implementation supports node-based and iteration-based animations. The software provides some algorithms for producing non-random initial layouts for Spring algorithms. File I/O using GML [19] has been implemented. In addition, users of LayoutShow applet can choose to perform local file I/O since LayoutShow is a signed applet. To our knowledge, LayoutShow is the first graph drawing software with this feature.

In this Chapter, we first discuss LayoutShow's features in more detail. Then, we describe the way different components of the system interact with each other, and

introduce each individual package in the system and elaborate on the functionalities of their classes. Finally, a list of known bugs and limitations are presented.

LayoutShow has been implemented using JDK 1.1.6 [34] under Solaris 2.5.1 (SunOS 5.5.1). It has also been tested in Linux 2.0.36, Windows 98, and Windows NT 4.0 as an application as well as an applet. The source, bytecode, and API of LayoutShow classes, and the LayoutShow applet can currently be found at "http://www.cs.yorku.ca /~lila/work.html".

4.2 Features

4.2.1 Outline

LayoutShow provides facilities to

- Generate a variety of graphs: complete, rectangular, hexagonal, triangular, complete binary tree, random, hypercube, and circular.
- Draw graphs using different force-directed Spring algorithms: CostSpring (see Section 3.3), GEM [15], FR [16], KK[21], and a combination of eigenvectors layout [29] and CostSpring (see Section 4.4.2).
- Generate initial layouts for Spring algorithms using: CostChosen (see Section 3.3), Insert [15], and EigenLayout [29].
- Randomize a graph.

- Obtain graph quality measurements[1]: number of edge crossings, longest edge to shortest edge ratio, edge length deviation, and cost value (see Section 3.2.4).
- Perform iteration-based and node-based animations (see Section 4.4.4).
- Configure the algorithms.
- Label the nodes.
- Reading/Writing a graph from/to disk using the GML [19] file format.

4.2.2 The LayoutShow Applet

We mentioned that LayoutShow is an application and also an applet. When running LayoutShow as an applet, all the classes that LayoutShow needs are down-loaded in one Internet transaction. This has been achieved through the use of a JAR, Java Archive file [10]. A JAR file can contain the classes and auxiliary resources that an applet uses. By using this, there will be no need for opening a new connection when a new class must be loaded for the applet. This may slightly slow down the speed at which the applet is loaded, but it will certainly speed up the execution once the applet is loaded and is running. A JAR file can also be compressed, as the JAR file for LayoutShow is. The current size of this file is about 0.5 megabytes.

Due to Java security restrictions a regular applet cannot access files on the local disk[2]. For this reason, VGJ [24], another graph drawing tool that is available as an applet, does not allow load and save operations. Although this restriction exists for

[1]These quality measurements are mainly relevant to the layouts produced by a force-directed spring-based algorithm which LayoutShow focuses on.

[2]Any discussion on Java security in this thesis refers to the Java 1.1.x security model [10, 34].

applets by default, but Java has provided ways for permitting an applet to access a local disk and have all the privileges that a Java application has. This is done using *signing and verifying* JAR files. We have signed the JAR file of LayoutShow, providing the option of using the LayoutShow applet with all the privileges of an application including File I/O. This is particularly important since traditionally graph drawing applets such as VGJ did not allow file I/O that is an important part of graph visualization[3]. GDS [8] requires the user to send his/her data files to their server, and they return the URL of the file that contains the graph layout (produced on the server side) to the user. This approach is slow especially if the graph is large. In addition, with the increasing speed of the processors that average users currently have, there is no need for relying on a server to do the computations when the computation can be done locally and reasonably fast.

The details of the concepts and procedures of signing and verifying an applet can be found in [10]. Therefore, we only describe these briefly. Electronically signing an applet is the same as signing a paper by pen. Anyone who recognizes our signature can trust our applet and grant some special privileges to it. Recognizing the electronic signature of an applet is called verification. The process of signing an applet involves a signer to sign the JAR file of the applet using a private key, and place the corresponding public key and its certificate (i.e. a digitally signed statement from trusted firms specializing in digital security) in the JAR file. These can be done using the *javakey* tool provided by JDK [34]. On the user side, the public key and

[3]This is because data may have been produced by another tool and saved in a file, and now the user needs to visualize this data.

the certificate can be obtained by the user from the signer in a file normally called `identitydb.obj`. Having this file, the user indicates to its Java-enabled web browser that it trusts a certain identity, and if the web browser loads an applet whose JAR file is signed by this identity it knows to grant privileges of an application to that applet.

4.3 The Overall Design of the System

In this Section we elaborate on the way that various components of the LayoutShow software interact with each other to support the computations for finding the new node positions, drawing the graph, and animation. The computation of node positions for a graph (computation module), and its actual drawing on the screen (display module) make up the two main modules of any graph drawing software. Traditionally in softwares such as Graphlet [20] and VGJ [24], the new positions of the nodes are first computed, and then the graph is (re)drawn. These two actions happen in a sequential order. We have used the multi-threading capabilities of Java for simultaneous location computations and drawing. To the best of our knowledge on the Java-based graph drawing softwares (see Section 2.4), this is a new design on the way these two modules cooperate in such tools. Figure 4.1 shows the relationship between the computation and the display modules.

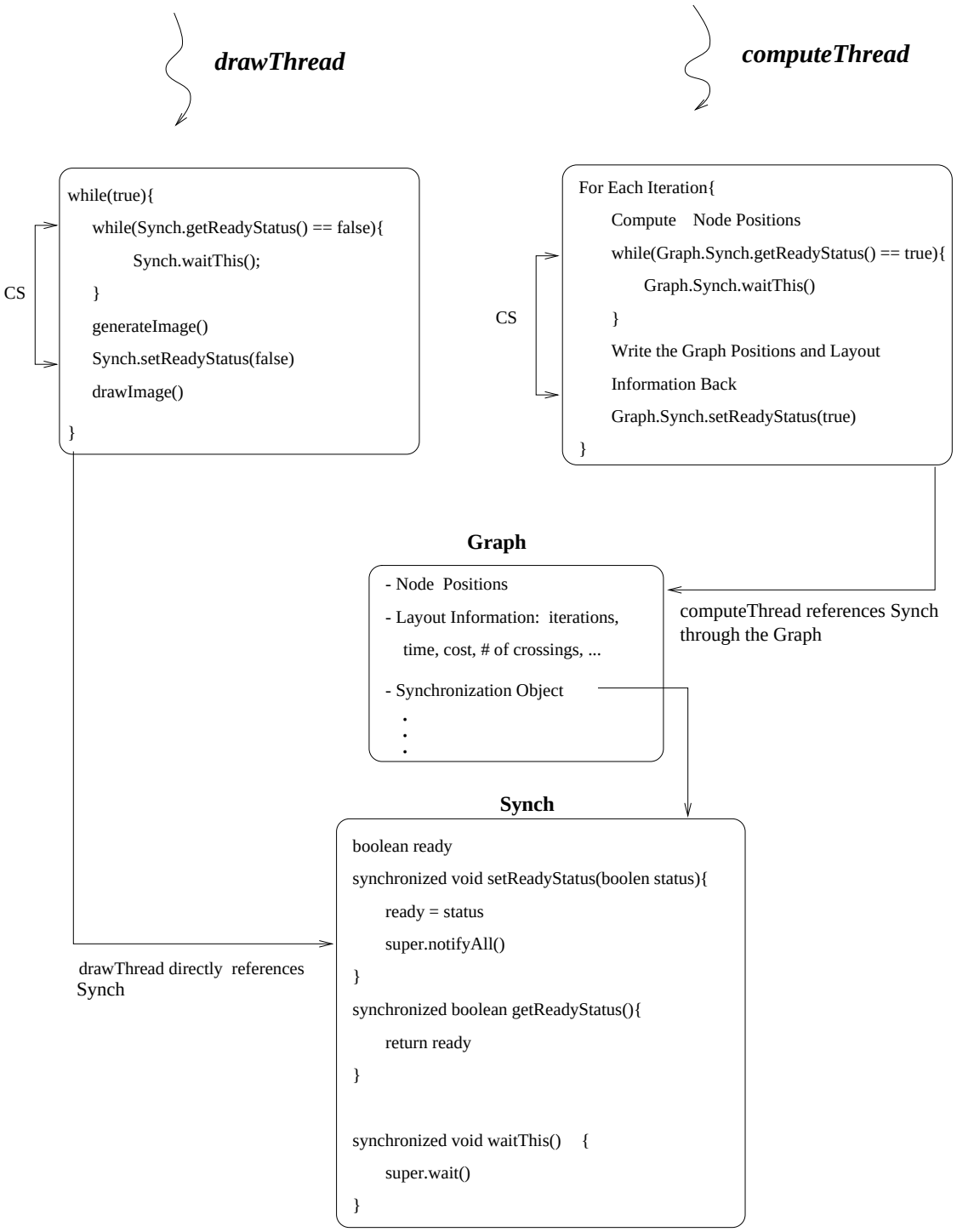

Figure 4.1: Synchronization scheme between `drawThread` and `computeThread` in LayoutShow.

The Two Threads

The tasks of computation and display are managed by two threads: `drawThread` and `computeThread` that share the graph as a common data. As a result, the segments of the code in which these two threads read or modify the graph are considered as critical sections, and must not be executed simultaneously[4]. Furthermore, the order in which these sections are executed is crucial.

The Critical Sections

The two critical sections are:

1. The writing back of the new node positions by the `computeThread`.

2. The reading of these new positions by the `drawThread`.

It is clear that item 1 above should be executed before item 2, and synchronization is required to manage this.

Synchronization

The synchronization of the critical sections are managed by using the wait and notify mechanisms of Java [23] through a synchronization object: `Synch` (see Figure 4.1). As we can see in this Figure, the `drawThread` loops infinitely, and in each iteration if the Synch object indicates that the graph is not ready to be drawn then the `drawThread` waits on the `Synch` object[5]. On the other hand, after writing the node positions,

[4]We must note that by simultaneous execution of threads we do not mean that threads run on multiple processors. What we refer to is the processor sharing by multiple threads through pre-emption.

[5]Object A waits on object B when object A calls the wait function of object B. Any call to notify or notifyAll of B can resume A.

the `computeThread` calls `Synch.setReadyStatus(true)` which in effect calls the `notifyAll` function of `Synch`. This resumes `drawThread` which now can start generating the image. At this point, the `computeThread` can continue computing the node positions for the next iteration. However, it cannot write the new positions back to the graph unless the `drawThread` has already called `Synch.setReadyStatus(false)`. If the `drawThread` has not called this function the `computeThread` will wait on the `Synch`. The `drawThread` makes this call after it reads the graph and generates the image, however, the actual drawing of the image occurs after the call to this function. As a result, the code segments that are labeled with CS (for critical section) in Figure 4.1 cannot be executed simultaneously by the two threads. And, the node positions are computed before the image is drawn. We must also note that the methods of `Synch` object are synchronized, and therefore, only one thread at a time may exist in this object.

If the animation option is off, the node positions are only written back once at the end of the algorithm by the `computeThread`. However, in case of iteration-based animation (for more on iteration-based and node-based animations, refer to Section 4.4.4) the node positions are all written back once at the end of each global iteration, and the image is redrawn. In case of node-based animation, the position of the node that has moved is written back and the image is redrawn. For any choice of animation, the computation of the node position(s) and generation/drawing of the graph can occur concurrently resulting in a smoother animation. We must note that the communication between the computation and drawing modules is one way. Precisely, the computation module provides the drawing modules with new

node positions, and there is no information passed to the computation module by the drawing module.

Remarks

There is only one `drawThread` per drawing `Canvas` while LayoutShow is running. However, a new `computeThread` is generated every time a graph drawing algorithm is instructed by GUI to execute and find the node positions of a graph. The `Synch` object is also unique and unchanged while LayoutShow is running. Due to the fact that the graph may be `null` at some point (i.e. there is nothing to be drawn), the `drawThread` has direct access to the `Synch` object. However, a `computeThread` can reference `Synch` through the graph. Each graph has a handle to the `Synch` that is assigned upon its generation.

The last remark that we like to make is that, the layout algorithms do not directly communicate with the classes of the GUI package. These algorithms receive the graph and a `boolean` flag indicating whether animation is required, and they find the node positions. The only way of communication with the drawing module (i.e. GUI) is through writing back (the node positions and other information such as time) into the graph, and through the synchronization mechanisim. This type of design allows for another module other than the drawing module to be in place of GUI. An example of such module is: one that would write the node positions found by an algorithm onto a disk drive.

4.4 Packages

In this Section, we examine the packages of LayoutShow. We list all the classes in each package along with their visibility modifiers [6], and review the important features that each package contributes to the tool. Figure 4.2 shows the relationship among the packages. We see that the `layoutstructs` package is used by all other packages, while `gui` is not used by any other package.

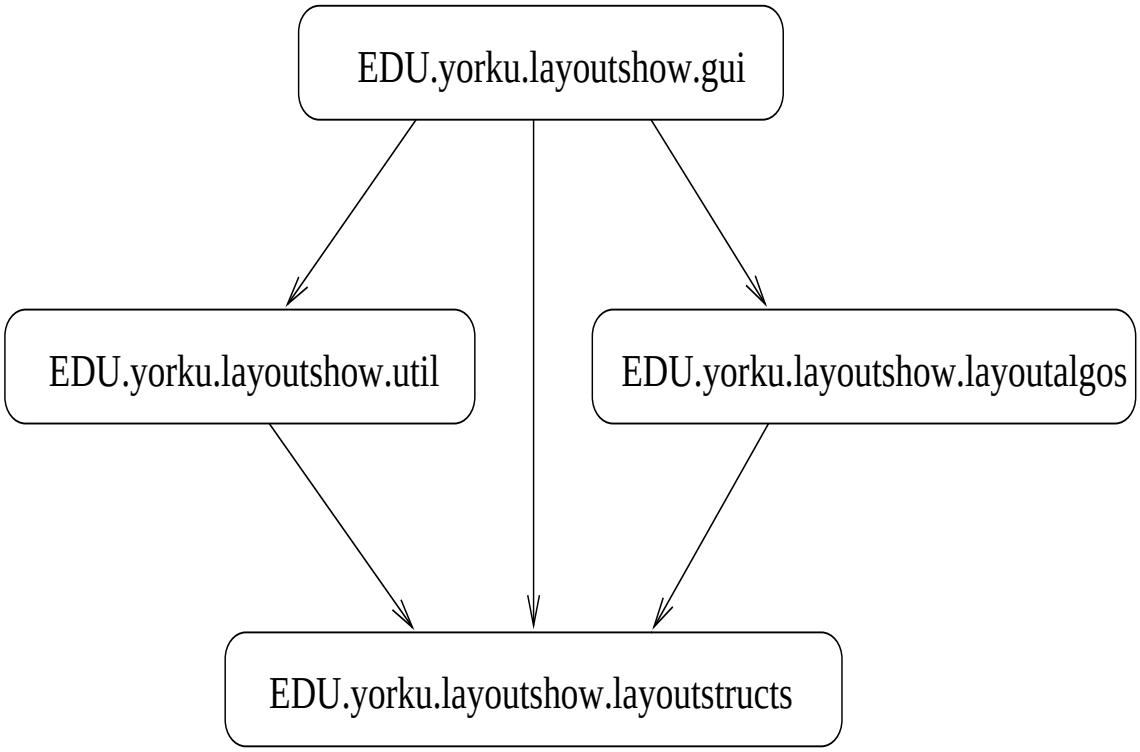

Figure 4.2: Usage of a package by other packages in LayoutShow. For instance, the arrow from `gui` package to `util` package indicates that `gui` makes use of one or more classes of `util`.

[6] Visibility modifiers are keywords that, when applied to a class, indicate where a class is visible. A `public` class is visible by every other class, and a `protected` class is visible by its subclasses as well as other classes in its package [9].

class	*Visibility Modifier*
GraphPic	public abstract
VectorGraphPic	public
SynchronizeGraph	public
VertexPic	public
EdgePic	public
LayoutStructException	public
VertexExistException	public
VertexNotExistException	public

Table 4.1: Classes of `layoutstructs` package with their visibility modifiers.

4.4.1 Graph Structures: EDU.yorku.layoutshow.layoutstructs

This package provides the data structures that are required to hold the adjacency information of a graph as well as its layout information. Table 4.1 shows all the classes in this package.

GraphPic and SynchronizeGraph

A graph class is one of the main components of any graph drawing system. `GraphPic` is an abstract class that is the superclass of any graph class used in LayoutShow system. It supports methods to build a graph and to obtain information about a graph including: adding edges and vertices, returning an enumeration of edges or vertices, and obtaining a vertex given its number. This class has been chosen to be an abstract class to support different implementations of the graph class in LayoutShow. This is useful in expanding the system in the future. For instance, if the current implementation of `GraphPic` in LayoutShow (i.e. `VectorGraphPic`) is not efficient for a new

drawing algorithm, then a new and more suitable class may extend `GraphPic`. As long as the functionality of this new class conforms to the specifications of the abstract functions of `GraphPic` the current classes would work fine with this new class. In this case through reflection in Java, minimal to no code modification will have to be made. Reflection allows for creating an instance of a class whose name is not known until runtime [10]. In the current implementation, there is a `String` variable that holds the name of this class that is currently "EDU.yorku.layoutshow.VectorGraphPic". Any instance of this class is created using this `String` and the reflection API of Java. If a new class is to be used in place of `VectorGraphPic` the value of this string is the only change that is required to be made.

`GraphPic` also has a reference to a `SynchronizeGraph` object that has been further explained in Section 4.3 under the variable name `Synch`.

VectorGraphPic, VertexPic, and EdgePic

`VectorGraphPic` implements `GraphPic`. This class makes use of `VertexPic` and `EdgePic` classes. Figure 4.3 shows the structure of `VectorGraphPic` and its relationship with vertex and edge classes for a small directed graph with 3 nodes. A vector holds the references to vertices of the graph. For fast look up of a vertex, a hash table is used to map the id of a vertex to its index in the vector. `VectorGraphPic` has an attribute that indicates whether a graph is directed or not. If the graph has one directed edge, then it is considered to be directed.

`VertexPic` is the data structure of each vertex. It includes layout and adjacency information about a vertex. The layout information includes: (x, y) coordinates,

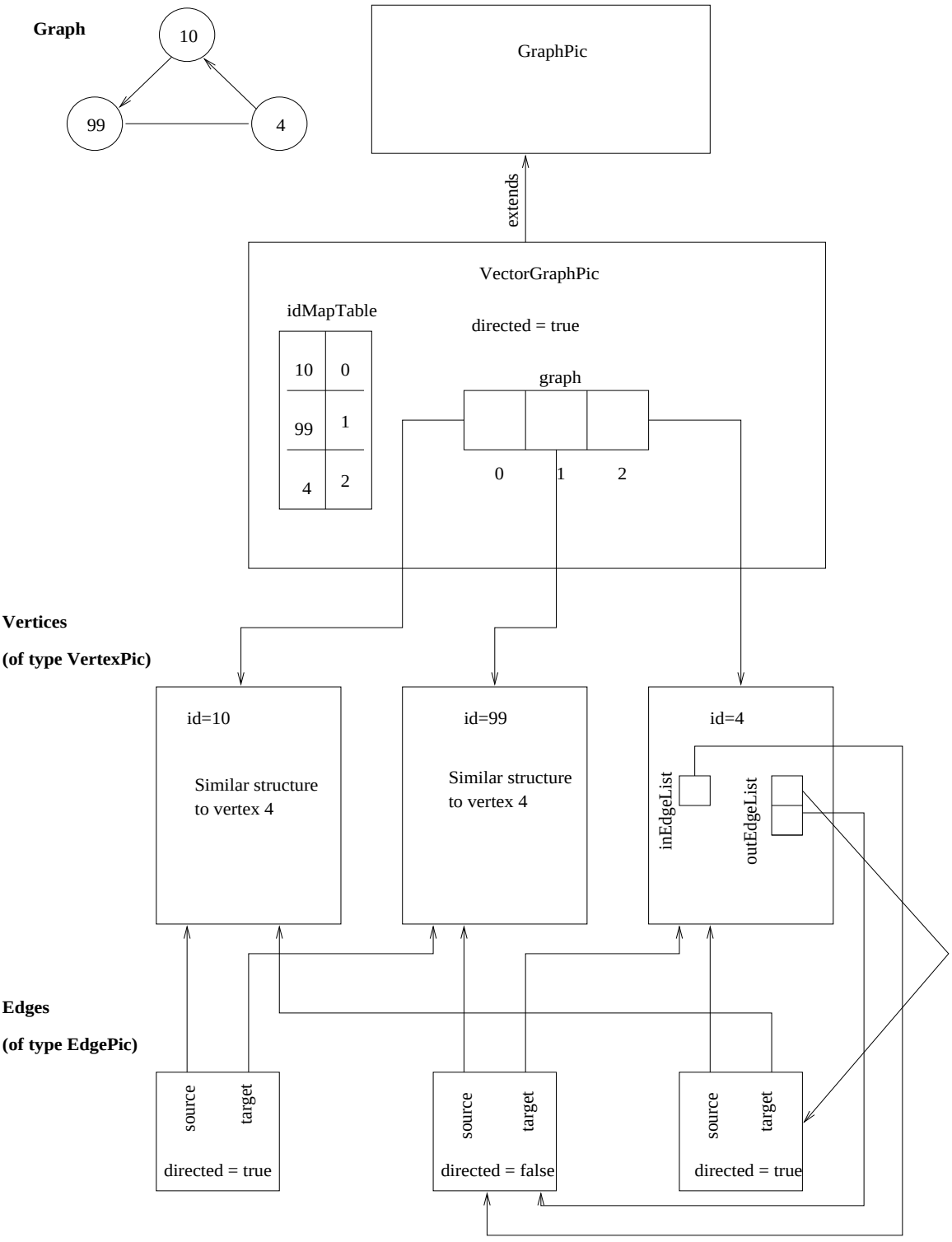

Figure 4.3: The internal structure of a `VectorGraphPic` object.

label, shape, width, height, and color. The adjacency information includes: a list of edges going out of the node, and a list of edges entering into a node. An undirected edge appears in both of these lists. To experiment with JGL, The Generic Collection Library for Java [27], we have implemented the `outEdgeList` and the `inEdgeList` using the `DList` class (doubly-linked list class) of JGL.

`EdgePic` holds the layout and adjacency information of an edge. Currently, color is the only layout information that `EdgePic` contains. In addition, references to source and target and an attribute that indicates whether an edge is directed or not make up the information of an edge. In the case of an undirected edge, the terms source and target are not meaningful and these variables are arbitrarily assigned to the two end-point vertices of an edge.

4.4.2 Layout Algorithms: EDU.yorku.layoutshow.layoutalgos

In this Section, we review the package that contains the algorithms that are used to layout graphs. Table 4.2 presents the classes of this package.

SpringAlgorithms

The `SpringAlgorithms` class provides the only public interface of `layoutalgos` package. This class consists of static data members and methods. The data members are the configurable attributes of the algorithms in this package such as the maximum number of iterations. The static methods are used to run a certain algorithm. They take the following general form

class	*Visibility Modifier*
SpringAlgorithms	public
SpringLayout	package abstract
CostOrientedLayout	package
GEMLayout	package
KK	package
GraphLetFR	package
EigenLayout	package
GraphQuality	package
SpringGraph	package abstract
SpringVertex	package
CostGraph	package
CostVertex	package
GEMGraph	package
GEMVertex	package
LayoutAlgoException	public
NullGraphtException	public

Table 4.2: Classes of `layoutalgos` package with their visibility modifiers.

```
public static Thread algorithm(GraphPic graph,
                               boolean animation,
                               boolean oneNodeAtATime,
                               int initialization)
```

where algorithm is a layout algorithm such as GEM. The `animation` , `oneNodeAtATime` (i.e. node-based animation), and `initialization` parameters are present if they are applicable to the algorithm. For instance, `oneNodeAtATime` is not meaningful in the context of FR algorithm where all the nodes are repositioned at once. Therefore, `oneNodeAtATime` is not present as a parameter of the static method that runs the FR algorithm.

In the body of these static methods, an instance of the appropriate algorithm is created, and its `findLayout` method is called. This method starts the thread that carries out the execution of the algorithm. This thread is returned to the user of the package in case termination, suspension, or resumption of this thread will be required. The only exceptions are `graphQuality` and `costChosenInitialLayout`. These two methods do not return a thread. In case of `graphQuality`, the same thread that calls this method computes the graph quality values. This is acceptable since the computation of quality values is often fast. `costChosenInitialLayout` method spawns a number of threads (by creating instances of `CostOrientedLayout`), each of which is responsible for finding the layout of the graph starting from a different initial random layout. The calling thread of this method waits for these threads to join, and then returns the index of the resulting layout with the smallest cost value. As a result, the calling thread of this method should not be a thread such as the thread that handles the GUI events, or it will be blocked until this method returns.

SpringLayout, CostOrientedLayout, GEMLayout, KK, and GraphLetFR

`SpringLayout` is the superclass of all the Spring algorithms in this package. It is an abstract class, and contains a number of common attributes and methods for Spring algorithms. The classes that implement various Spring algorithms in LayoutShow are

- `CostOrientedLayout` which implements *CostSpring*: the Spring algorithm introduced in this thesis (see Section 3.3).

- `GEMLayout` which implements the *GEM* algorithm [15].

- `KK` which implements the *KK* algorithm [21].

- `GraphLetFR` which implements the Graphlet [20] version of the *FR* algorithm [16] in Java.

An instance of each of these classes spawns a thread that carries out its computations. These classes create their own internal data structures for a graph and its vertices. The reasons for implementing these internal data structures will be described at the end of this Section. In addition, before the end of the algorithm or in each iteration of the algorithm (if animation is requested), the (x, y) coordinates of node(s) along with other information such as iteration count and time are written back to the global graph (see Section 4.3).

EigenLayout

To introduce an alternative graph drawing approach to Spring algorithms, we have implemented the eigenvector layout algorithm. The details of this method of graph layout has been presented in [29]. `EigenLayout` class uses JSci [18], a science and mathematics API for Java, to find the eigenvectors of the Laplacian matrix of a given graph. In some cases, this method of graph drawing finds layouts with overlapping nodes. Therefore, this class allows the combination of eigenvectors layout with Spring algorithm. This means that the user of the class may choose to apply a Spring algorithm (in this case CostSpring) to the resulting layout of eigenvectors algorithm if this layout has overlapping nodes.

GraphQuality

The `GraphQuality` class provides methods to find each of the following quality values of a graph:

- number of edge crossings,
- longest edge to shortest edge ratio,
- edge length deviation,
- cost value (see Section 3.2.4).

In addition, this class provides a method which finds all these quality values and writes them back into the graph to be used by the `gui` package. These quality values are particularly meaningful for the layouts generated by the family of Spring algorithms.

SpringGraph, SpringVertex, CostGraph, CostVertex, GEMGraph, and GEMVertex

In this package, we have implemented local graph and vertex structures for different Spring algorithms. There are two reasons for implementing these. First, most Spring layout algorithms requires their own unique attributes for each vertex. For instance, CostSpring stores the current and previous cost values for each node, and a GEM vertex has information such as mass, impulse, temperature, and direction. Secondly and more importantly, in the design and implementation of LayoutShow we have tried to keep the efficiency of the algorithms independent of the implementation of

the global graph structure. This means that in the future, in the case of changing the implementation of `GraphPic` it is not required that this new implementatioin work efficiently with currently existing Spring algorithms. This is due to the fact that the global graph structure is copied into a local and efficient one that is used by Spring algorithms.

`SpringGraph` and `SpringVertex` are the super-classes of all the local graph and vertex classes respectively. The `SpringGraph` abstract class's data members are the global parent graph, and an array of `SpringVertex` objects. It also has the following abstract method

```
public abstract void writeBack(SpringLayout layout,
                               boolean oneNodeAtATime,
                               int node);
```

Method `writeBack` writes back the node positions and layout information to the global graph. Any local vertex is a subclass of `SpringVertex` which holds the indices of adjacent and non-adjacent vertices of a node in the array of `SpringVertex` objects in the `SpringGraph` object.

`KK`, `GraphLetFR`, and `CostOrientedLayout` use `CostGraph` and `CostVertex`, and `GEMLayout` uses `GEMGraph` and `GEMVertex`.

4.4.3 Utility: EDU.yorku.layoutshow.util

The `util` package is a small package consisting of two classes as shown in Table 4.3. These classes support graph input/output from/to files and graph generation. We

class	*Visibility Modifier*
GML	public
GraphGenerator	public

Table 4.3: Classes of `util` package with their visibility modifiers.

note that all methods of these two classes are static. Below we briefly describe each of these classes.

GML

This class reads and writes a graph from and to a file in GML [19] file format. The `GML` class provides the following two static methods

```
public static GraphPic read(FileInputStream fis, String graphClass)
public static void write(PrintWriter pw, GraphPic g)
```

where `read` inputs a graph from a file associated with `fis` into a graph of type `graphClass`, and returns this graph. The `graphClass` parameter indicates the exact type of the global graph which is currently `VertexGraphPic` (see Section 4.4.1). This `String` is used along with reflection API of Java to create an instance of the class indicated by `graphClass`. Similarly, the `write` method writes out the graph `g` into a file associated with `pw`.

Currently the `read` function can input the following attributes: a node's id, label, (x, y) coordinates, its width and height, color, and type. The types that are supported are oval and rectangle. For an edge this method can input: its source, target, color, and whether it is directed or not. Except for a node's id and an edge's source and

target ids, the rest of the attributes are optional. Default values will be used for these attributes if the user does not specify their values.

The `write` method writes the non-optional attributes of nodes and edges mentioned above into a file, along with the optional attibutes whose values are not the default values.

GraphGenerator

This class contains a number of static methods of the following form

```
public static GraphPic graphType(String graphClass, ...)
```

where `graphType` indicates the type of the graph that is to be generated, for instance, a complete graph. `graphClass` is the exact type of the global graph which is currently `VertexGraphPic` (see Section 4.4.1). '...' is in place of argument(s) that may differ for one graph type to another. For instance, it is the number of nodes for a complete graph, and the number of rows and columns for a rectangular grid. This method returns the graph that has been generated. Note that the (x, y) coordinates of the vertices of a graph generated with this method are not initialized. The following is the list of graph types that can be generated by the methods of `GraphGenerator` class

- Complete graph.
- Triangular grid.
- Five-point operator grid.

class	*Visibility Modifier*
LayoutShow	public
MainWindow	package
Menus	package
GraphFile	package
TopPanel	package
SouthPanel	package
InputWindow	package
CostConfigureWindow	package
AnimationWindow	package
QualityWindow	package
ErrorWindow	package

Table 4.4: Classes of `gui` package with their visibility modifiers.

- Hexagonal grid.
- Complete binary tree.
- Circular graph.
- Random graph.
- Hypercube.

4.4.4 Graphical User Interface: EDU.yorku.layoutshow.gui

In this Section, we provide an overview of the `gui` package of LayoutShow. Table 4.4 lists the classes in this package.

LayoutShow

`LayoutShow` class starts LayoutShow as an applet as well as an application. If this class is run as an application, then it will open the main window of LayoutShow. In case of an applet, this class creates a button that when pressed, a LayoutShow window will open up.

MainWindow, TopPanel, and SouthPanel

`MainWindow` extends AWT `Frame` [9], and has the following major components: a `Menu` object, a `TopPanel` object, an AWT `Panel` added to an AWT `ScrollPane`, and a `SouthPanel` object. Figure 4.4 shows a snapshot of an instance of `MainWindow` where its middle panel is a single `Canvas`.

`TopPanel` class extends the AWT `Panel` and is meant to hold a number of buttons who are needed frequently. Currently this panel holds a button to shuffle a graph, and stop, suspend, and resume buttons to manipulate the thread that is executing the algorithm.

`SouthPanel` class also extends the AWT `Panel` and has two AWT `TextField` objects to show the time that an algorithm takes to find the current layout, and the number of iterations.

The middle panel of `MainWindow` has `CardLayout` manager which allows for multiple `Component` objects to overlap [42]. This middle panel has two overlapping components: the default is a `Canvas` (see Figure 4.4), and the other one is a `Panel` with 10 `Canvas` objects (laid out in 5 columns and 2 rows) to support the simultaneous execution of Spring algorithms on different initial random layouts of a graph in

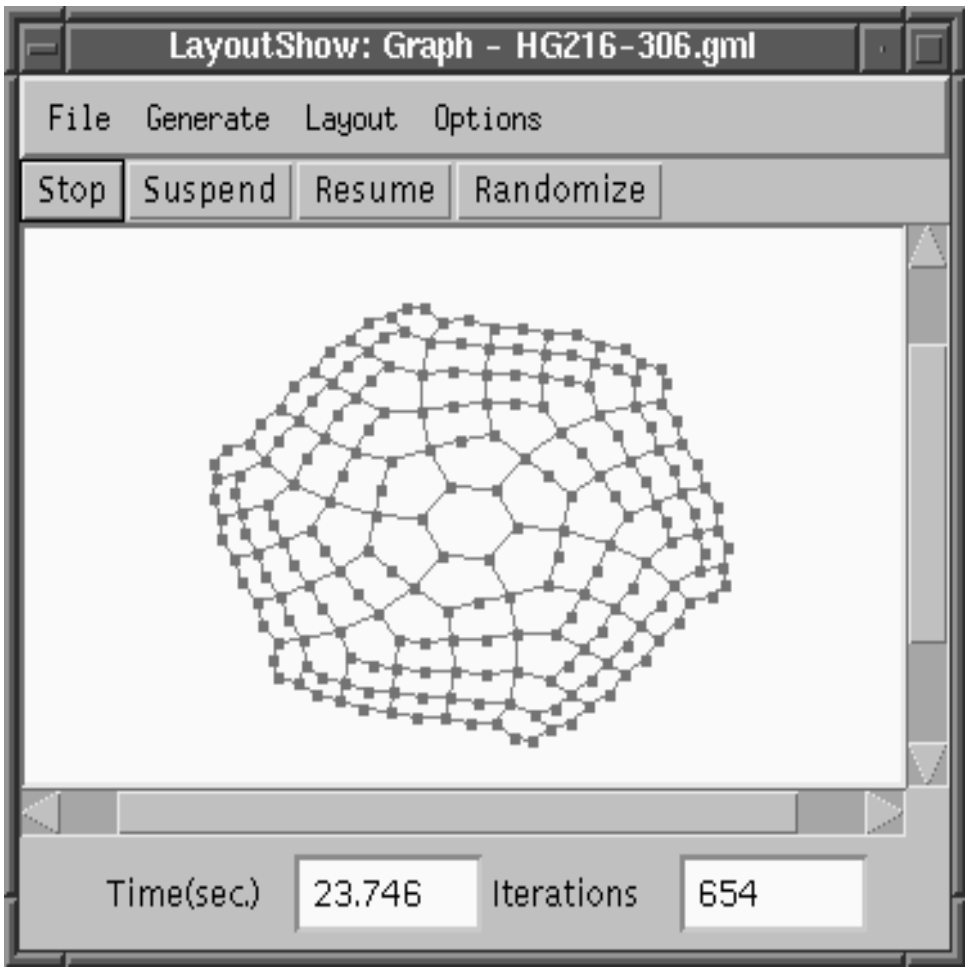

Figure 4.4: Snapshot of LayoutShow's main window.

CostChosen initial layout (see Section 3.3). The user can choose the number of initial layouts that are used where the maximum number of initial random layouts is 10. Once the algorithms for all initial layouts terminate, then the graph layout quality value of the resulting layouts will appear on top of each canvas with the lowest one flashing in red (see Section 3.3 for this quality value's formula). Then after a delay, the resulting layout with the lowest quality value will appear on the single canvas of

`MainWindow`. A snapshot of `MainWindow` with this multiple-canvas panel is shown in Figure 4.5.

Each canvas in the middle panel, draws an image using double buffering of Java AWT [42]. This results in the elimination of partial screen updates and flickering.

Menus

The `Menu` class extends the AWT `MenuBar`. As we can see in Figure 4.4, this class has four menus: File, Generate, Layout, and Options. The following is a list of the items in each of these menus:

- File: New, Open, Save, SaveAs, Print, Close, Quit.
- Generate: Complete Graph, Rectangular Grid, Hexagonal Grid, Triangular Grid, Tree, Random, Hypercube, Circle.
- Layout: Randomize, Quality, Generate Initial Layout, CostSpring, EigenSpring, FR, GEM, KK.
  - Generate Initial Layout: Insert, CostChosen, EigenLayout.
- Options: Animation, Configuration, Labeling.
  - Configuration: Insert, Cost Chosen, CostSpring, FR, GEM, KK.

GraphFile

`GraphFile` extends AWT `File`. An instance of this class is associated with any graph that is present on the single canvas of `MainWindow`. This class manages the clean and

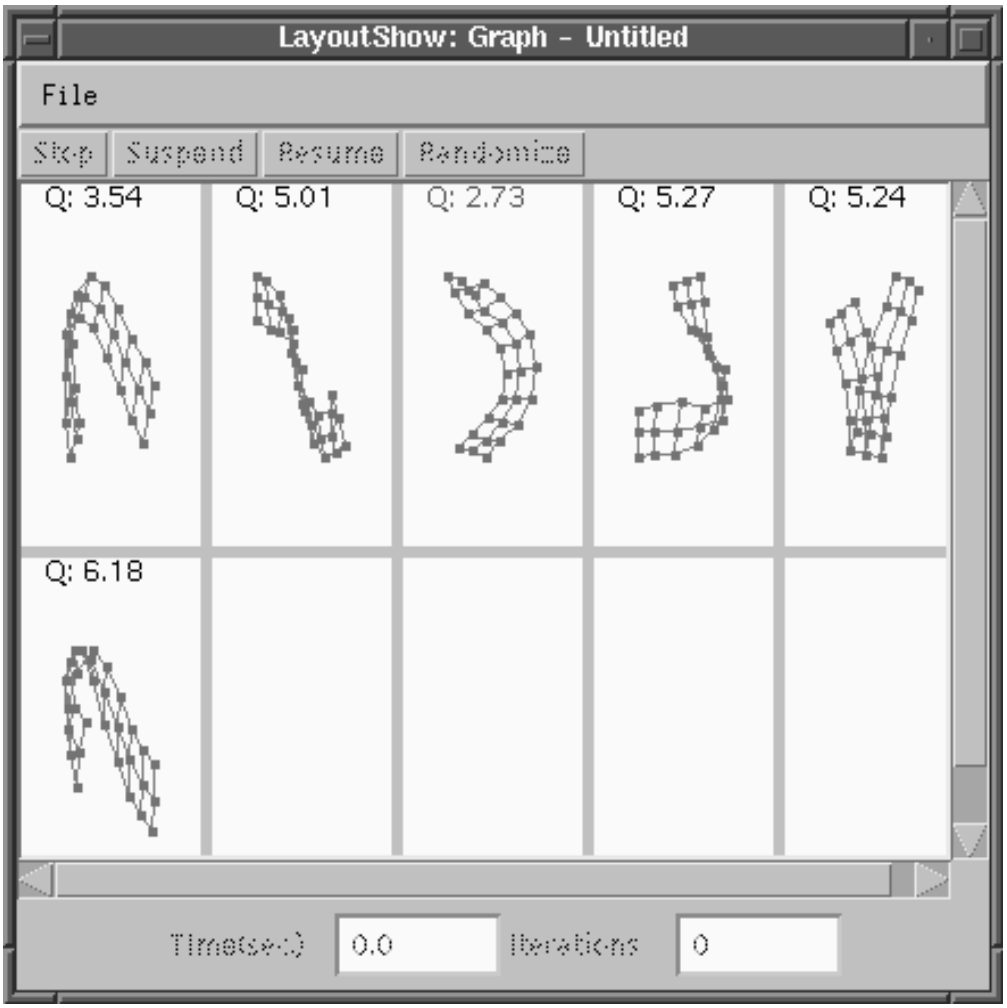

Figure 4.5: Snapshot of LayoutShow's main window with a multiple-canvas panel.

dirty state of a graph as well as the actions that need to be taken when New, Open, Save, and SaveAs menu items are invoked.

The Remaining Classes

The remaining classes are: `InputWindow`, `CostConfigureWindow`, `AnimationWindow`, `QualityWindow`, and `ErrorWindow`. With the exception of `QualityWindow` which extends AWT `Frame` all these classes extend AWT `Dialog`. The following is a brief description of each of these classes:

- `InputWindow`: inputs the information that is needed for graph generation. For example, for a complete graph this window will input the number of nodes.
- `CostConfigureWindow`: inputs some of the configurable attributes of Cost-Spring algorithm.
- `ErrorWindow`: displays an error message.
- `AnimationWindow`: allows the user to choose no animation, iteration-based animation, or node-based animation. We explain these further below.
- `QualityWindow`: displays the quality values of a layout (see Section 4.4.2).

Some of the Spring algorithms find the forces acting on a node and reposition this node. In this case, it is possible to update the layout after a node is repositioned, node-based animation, or after all the nodes are repositioned once, iteration-based animation. An instance of `AnimationWindow`, gives the user the opportunity to choose one of these animation types, or no animation at all. For Spring algorithms

such as FR [16] which moves the nodes at once, only iteration-based animation is applicable.

4.5 Bugs and Limitations

There are a number of known bugs and limitations in the current implementation of LayoutShow:

- Configuration windows for configuring Insert, FR, GEM, and KK have not been implemented. Currently, the default configurations for these algorithms are used.
- A very simple approach to labeling has been used which results in overlapping labels in some cases.
- Even though, `GraphPic` and `VectorGraphPic` classes support directed graphs, the drawing of directed edges have not been implemented in LayoutShow.
- Currently, Netscape and Internet Explorer, the two commonly used Internet browsers, can only support Java signed applets [10] if they have a Java Plug-in [35] installed. Our LayoutShow applet is signed by JDK 1.1.6 [34], and tested using Java Plug-in 1.1 [33]. Although, Sun has promised that the final version of Java Plug-in 1.2 would also support the applets signed by JDK 1.1.x, but we have not tested our signed applet using Java Plug-in 1.2.
- The animation in a LayoutShow applet that is running under the default Java virtual machine of a Netscape browser sometimes hangs. We recommend using

a Java Plug-in [33].

Chapter 5

Experimental Results

5.1 Outline of the Experiments

Our experiments are divided into two categories: measuring the efficiency of CostSpring algorithm, and evaluating the quality of its resulting layouts. We compare the running-time and the layout qualities of CostSpring with those of GEM [15] and FR [16] algorithms. Himsolt et al.'s studies [7] reported that among the family of force-directed algorithms, FR is fast for small graphs (with less than 60 nodes and edges), and GEM and their modified version of KK [21] are competitive in speed and outperform the others for both small and large graphs[1]. We did not use KK for comparison, since the original implementation of KK is very inefficient due to the large amount of computation that it requires before relocating a node, and the modified version of this algorithm (used in Himsolt et al.'s studies) produces drawing of planar graphs with many edge crossings. We confirm Himsolt et al.'s results that GEM is generally faster than FR for graphs with less than 180 nodes and edges. However

[1]The largest graph that was used by Himsolt et al. had a total of 180 nodes and edges.

according to our experiments, we do not confirm the results provided in [15] which indicate that GEM is faster than FR for graphs up to 255 nodes. See Appendix B for details of our experiment that shows this discrepancy.

Table 5.1 shows the test suite of 34 graphs that we have used in our experiments. This test suite is the same as what was used in [15] to test the GEM algorithm with the addition of graphs: 11, 12, 24, 25, 28, 32, and 34. To avoid bias, in all our experiments the average of the results of five runs for each algorithm are reported where five common initial random layouts of each graph are used for experimenting with all the three algorithms (unless noted otherwise).

In Section 5.5, we evaluate the running-time and the number of edge crossings of a combination of eigenvector and CostSpring algorithm that is implemented by LayoutShow. Moreover, in Section 5.6, we present the results of an experiment that evaluates the effectiveness of cost chosen initial layout (see 3.3) in reducing the chance of ending up in an unpleasing local minimum. Finally, we conclude this Chapter by an overall evaluation of the CostSpring, GEM, and FR in terms of running-time and layout quality based on the experiments presented in this Chapter.

5.2 Configuration

5.2.1 Hardware Configuration

All of our experiments (unless noted otherwise) are performed on a SUN Ultra-SPARC-IIi workstation with a 300MHz processor and 256 megabytes of RAM.

#	*Name*	$\|V\|$	$\|E\|$
Sparse			
1	Binary Tree 15	15	14
2	Path 16	16	15
3	Cycle 16	16	16
4	Star 24	24	23
5	Binary Tree 31	31	30
6	Path 48	48	47
7	Cycle 48	48	48
8	Binary Tree 63	63	62
9	Binary Tree 127	127	126
10	Path 128	128	127
11	Binary Tree 180	180	179
12	Cycle 220	220	220
13	Path 256	256	255
14	Binary Tree 255	255	254
Normal			
15	Wheel	13	24
16	4×4 Square Grid	16	24
17	Hypercube4D	16	32
18	Dodecahedron	20	30
19	Triangular Grid 28	28	63
20	Hypercube5D	32	80
21	7×7 Square Grid	49	84
22	Triangular Grid 55	55	135
23	Hexagonal Grid 96	96	132
24	5×20 Square Grid	100	175
25	10×12 Square Grid	120	218
26	Triangular Grid 120	120	315
27	Triangular Grid 210	210	570
28	Hexagonal Grid 216	216	306
29	16×16 Square Grid	256	480
Dense			
30	K12	12	66
31	K24	24	276
32	K50	50	1225
33	Hypercube6D	64	192
34	Hypercube8D	256	1024

Table 5.1: Test suite of 34 different graphs.

5.2.2 Software Configuration

The operating system that is used in our experiments is Solaris 2.5.1 (SunOS 5.5.1). We used the application version of LayoutShow under JDK 1.1.6.

GEM and FR implementations in LayoutShow are those of Graphlet [20] ported from C++ to Java. In addition, the default configurations of these algorithms in Graphlet [20] are used in our experiments. Refer to Appendix D for these configurations along with the default configuration of CostSpring used in our experiments.

5.3 Efficiency of CostSpring Algorithm

In this Section, we start by measuring how fast the CostSpring algorithm finds a layout or converges compared to GEM and FR algorithms. Then we provide the results of an experiment that measures the average number of sub-iterations of a node in phase 1 of CostSpring algorithm. We continue by presenting the results of an experiment that evaluates the effectiveness of our method of stopping the CostSpring algorithm. Finally, we provide data that show the overall performance of CostSpring, GEM, and FR for our test suite of graphs.

5.3.1 Convergence

In this Section, we run CostSpring, GEM, and FR on our test graphs with animation on, and for each graph record the iteration count at which the graph layout cannot be significantly improved further. For each graph and each algorithm, Figures [5.1-5.3] show the number of iterations at which the layout is found. We allowed GEM and

FR to run up to 1000 iterations (this is higher than the default maximum iteration count for GEM). We must emphasize that some graphs were still improving at 1000 iterations. In such cases, we report their convergence at 1000 iteration. From these Figures we can see that CostSpring outperforms FR in all cases. For graphs with less than 96 nodes (i.e. graphs: 1-8, 15-22, and 30-33), CostSpring and GEM find a layout using approximately the same number of iterations. However, for graphs with more than 96 nodes CostSpring finds a layout earlier than GEM with the exception of graph 34.

5.3.2 On Moving a Node More Than Once Per Iteration

The running-time of the inner do-loop of phase 1 of CostSpring algorithm (see Section 3.3) is not theoretically known. Thus, we conducted the experiment of this Section to find the average number of iterations of this loop for a node in one global iteration of the first phase of CostSpring algorithm. To find this average, A_G, for a node of a graph G of our test suite, we use the following formula

$$A_G = \frac{inner\_iterations_{phase1}}{|V| \times global\_iterations_{phase1}} \tag{5.1}$$

where $inner\_iterations_{phase1}$ is the total number of iterations of the inner do-loop of phase 1, and $global\_iterations_{phase1}$ is the total number of iterations of the outer do-loop of this phase. Table 5.2 shows the value of A_G for the graphs of our test suite. We can see that the value of A_G is below 2 for all graphs except for hypercubes. We have observed that hypercubes do not converge in the first phase of CostSpring

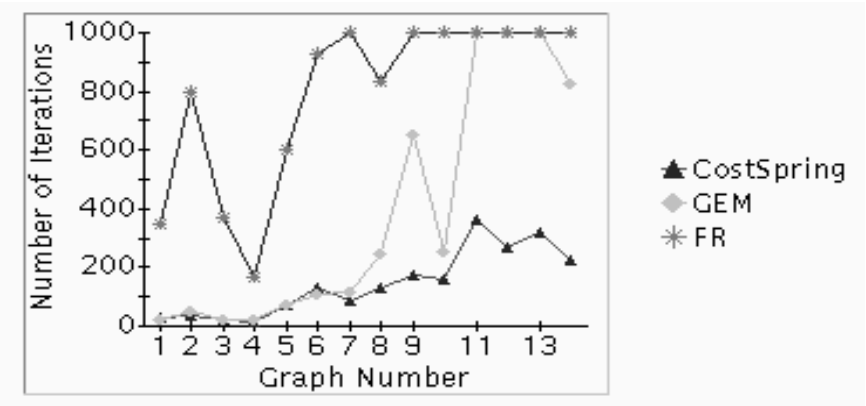

Figure 5.1: Number of iterations at which CostSpring, GEM, and FR converge for sparse graphs.

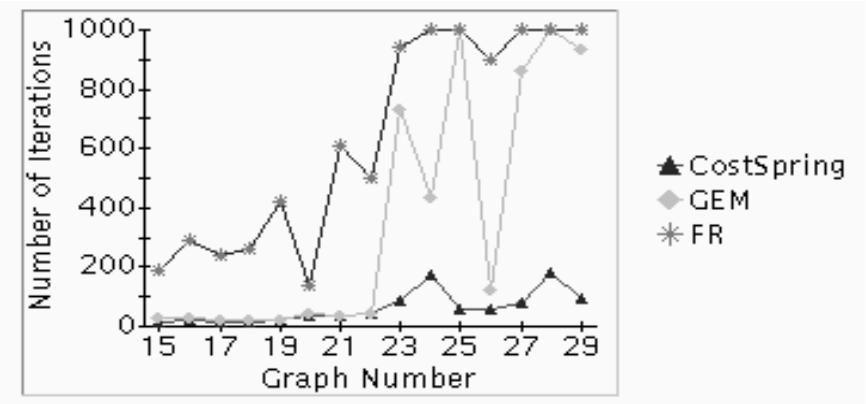

Figure 5.2: Number of iterations at which CostSpring, GEM, and FR converge for normal graphs.

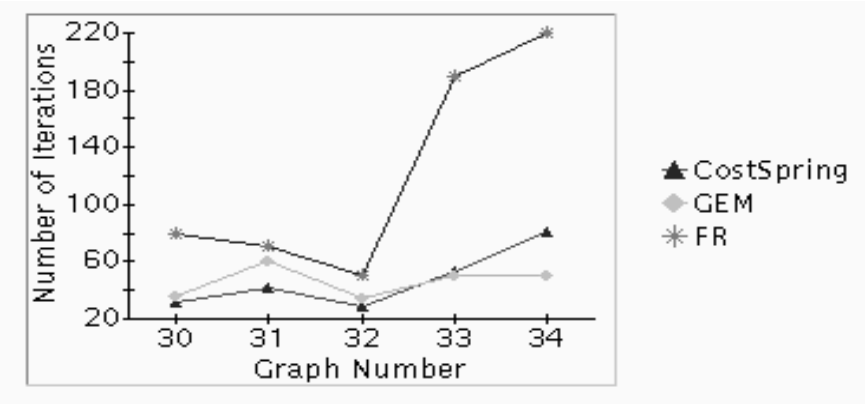

Figure 5.3: Number of iterations at which CostSpring, GEM, and FR converge for dense graphs.

algorithm. This results in larger number of movements for each node in each iteration of phase 1 of this algorithm.

5.3.3 Stopping

One of the challenges in current Spring algorithms is to find an effective method to terminate the algorithm once a layout has been found. In this Section, we conduct an experiment to show the effectiveness of our method for stopping the CostSpring algorithm (see Section 3.3). To achieve this, we run the CostSpring algorithm on our test suite of graphs, and for each graph we make a comparison between the iteration count at which a layout converges (the same data that was used in Section 5.3.1) and the iteration count at which the algorithm terminates. Naturally for a stopping method to be effective, these two values must be close, and a layout should converge before the algorithm terminates. Figures [5.4-5.6] demonstrate the result of this experiment. These Figures show acceptable results with the exception of a few large graphs. These are the cases in which oscillation occurs. Detection of oscillation can further improve this stopping method.

Table 5.3 shows that the stopping method of the Graphlet [20] version of GEM algorithm (see Section 2.2.3) did not work for any of the 34 graphs of our test suite. Note that the maximum number of iterations used in this implementation of GEM is $3 \times |V|$. The stopping method of Graphlet version of FR algorithm (see Section 2.2.2) only worked in some cases, and those cases were all for graphs with less than 50 nodes. Note that the maximum number of iterations used in this implementation

#	*Name*	A_G
1	Binary Tree 15	1.0602
2	Path 16	1.101
3	Cycle 16	1.1336
4	Star 24	1.0193
5	Binary Tree 31	1.0444
6	Path 48	1.0378
7	Cycle 48	1.0828
8	Binary Tree 63	1.027
9	Binary Tree 127	1.0248
10	Path 128	1.0549
11	Binary Tree 180	1.0258
12	Cycle 220	1.0355
13	Path 256	1.0547
14	Binary Tree 255	1.1364
15	Wheel	1.4434
16	4×4 Square Grid	1.4338
17	Hypercube4D	1.5729
18	Dodecahedron	1.325
19	Triangular Grid 28	1.6327
20	Hypercube5D	2.1172
21	7×7 Square Grid	1.3614
22	Triangular Grid 55	1.4939
23	Hexagonal Grid 96	1.3813
24	5×20 Square Grid	1.3644
25	10×12 Square Grid	1.3426
26	Triangular Grid 120	1.459
27	Triangular Grid 210	1.3892
28	Hexagonal Grid 216	1.3326
29	16×16 Square Grid	1.4373
30	K12	1.0083
31	K24	1.0042
32	K50	1
33	Hypercube6D	3.1914
34	Hypercube8D	4.5666

Table 5.2: Average number of sub-iterations of a node in phase 1 of CostSpring.

of FR is 1000.

5.3.4 Overall Performance

Table 5.3 presents the execution time of CostSpring, GEM, and FR in seconds along with the number of iterations of each algorithm[2] on the set of 34 graphs of our test suite. Figures [5.7-5.9] represents the time measurements of Table 5.3. Note that the default configuration of each algorithm has been used for this experiment (see Appendix D).

CostSpring shows better running-time performance than GEM and FR for all of our test graphs with the exception of graphs 14, and 24. In the case of these two graphs as mentioned in Section 5.3.3, our stopping method shows a weakness in detecting oscillation, and therefore, the execution time of CostSpring for graphs 14 and 24 can improve if this weakness is resolved. In addition, for graph 24 as we can see in Figure 5.2, 300 iterations of GEM is insufficient for its layout to converge.

5.4 Quality of the Layouts of CostSpring Algorithm

In this Section, we examine the quality of the layouts of CostSpring algorithm, and compare these layout qualities with those of GEM and FR algorithms. Our quality measurement criteria include the conventional ones: number of edge crossings, edge length deviations, and ratio of the longest edge to the shortest edge (used in [7]) as well as the cost value of the graph layout (see Section 3.2.4). Appendix C, contains

[2]The iteration counts for CostSpring refer to the number of global iterations (see Section 3.3).

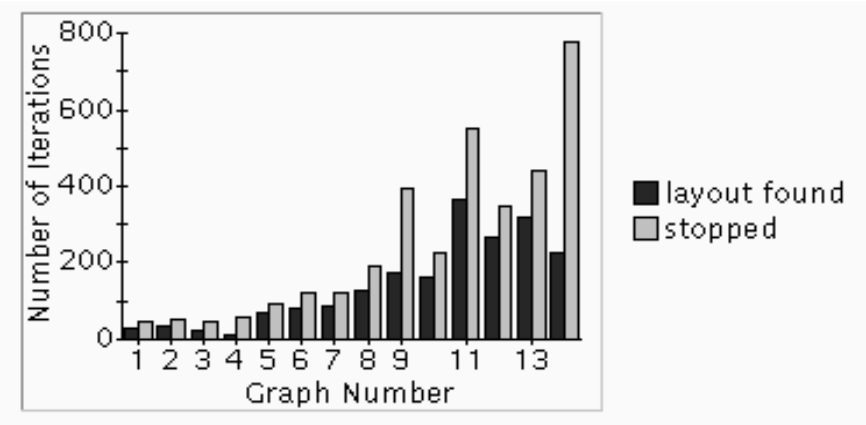

Figure 5.4: Number of iterations of CostSpring's convergence and stopping for sparse graphs.

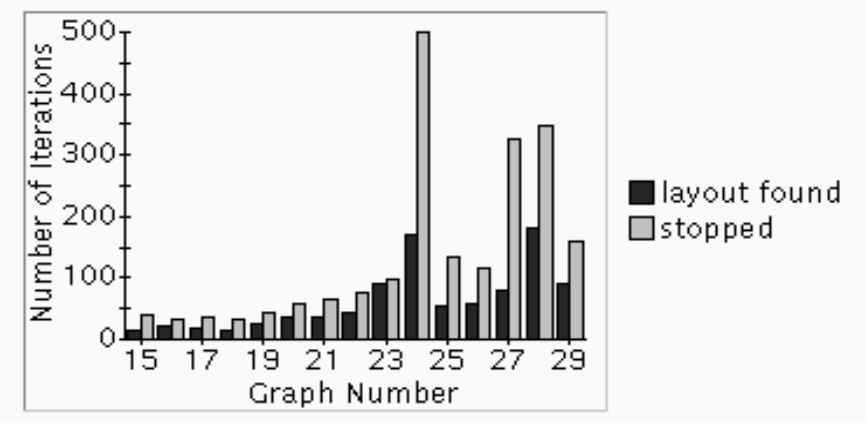

Figure 5.5: Number of iterations of CostSpring's convergence and stopping for normal graphs.

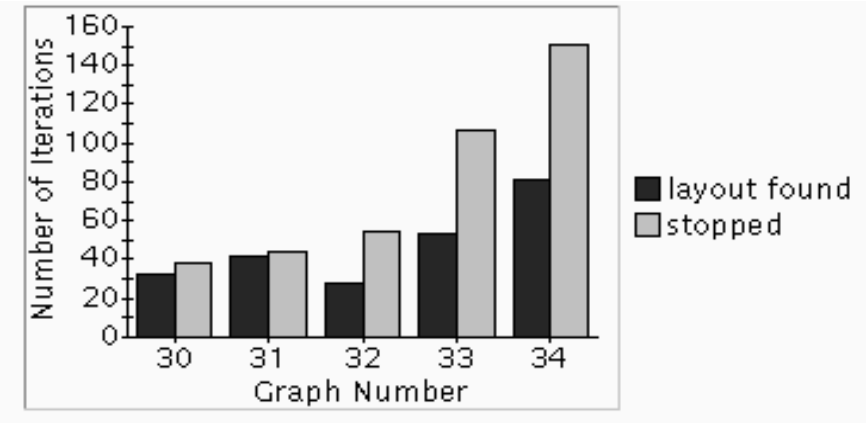

Figure 5.6: Number of iterations of CostSpring's convergence and stopping for dense graphs.

		CostSpring		GEM		FR	
#	*Name*	*iter.*	*time*[*s*]	*iter.*	*time*[*s*]	*iter.*	*time*[*s*]
1	Binary Tree 15	46.2	0.017	45	0.022	669.6	0.115
2	Path 16	53.6	0.018	48	0.022	1000	0.148
3	Cycle 16	45.6	0.015	48	0.022	561.8	0.083
4	Star 24	59.6	0.035	72	0.061	1000	0.271
5	Binary Tree 31	93.8	0.116	93	0.137	993	0.434
6	Path 48	123.2	0.239	144	0.354	1000	0.968
7	Cycle 48	120.8	0.255	144	0.375	1000	0.972
8	Binary Tree 63	189.2	0.644	189	0.734	1000	1.616
9	Binary Tree 127	392	4.517	381	5.307	1000	6.289
10	Path 128	224	2.723	384	5.548	1000	7.244
11	Binary Tree 180	552.6	12.71	540	14.29	1000	13.13
12	Cycle 220	345	12.312	660	26.51	1000	24.30
13	Path 256	441.6	21.88	768	43.27	1000	27.00
14	Binary Tree 255	779.2	42.09	765	44.41	1000	27.29
15	Wheel	41	0.015	39	0.019	239	0.034
16	4×4 Square Grid	33.2	0.013	48	0.024	320	0.052
17	Hypercube4D	36.8	0.015	48	0.025	212.6	0.036
18	Dodecahedron	32.2	0.017	60	0.04	349	0.074
19	Triangular Grid 28	42.6	0.042	84	0.095	497.2	0.2
20	Hypercube5D	58.2	0.075	96	0.137	308.4	0.155
21	7×7 Square Grid	64.6	0.162	147	0.395	714.2	0.726
22	Triangular Grid 55	75.8	0.213	165	0.542	1000	1.416
23	Hexagonal Grid 96	96.2	0.696	288	2.802	1000	3.767
24	5×20 Square Grid	500	3.854	300	2.743	1000	4.036
25	10×12 Square Grid	135	1.690	360	4.456	1000	6.305
26	Triangular Grid 120	116.6	1.528	360	4.47	1000	6.11
27	Triangular Grid 210	325	14.16	630	23.14	1000	18.69
28	Hexagonal Grid 216	347	14.919	648	24.70	1000	19.94
29	16×16 Square Grid	159.8	10.38	768	43.67	1000	28.12
30	K12	38.4	0.014	36	0.018	813.6	0.137
31	K24	43.6	0.039	72	0.106	1000	0.573
32	K50	54	0.191	150	0.754	1000	2.532
33	Hypercube6D	106.8	0.451	192	0.887	1000	1.766
34	Hypercube8D	150.8	18.01	768	42.11	1000	28.95

Table 5.3: Running times and iteration counts for CostSpring, GEM, and FR.

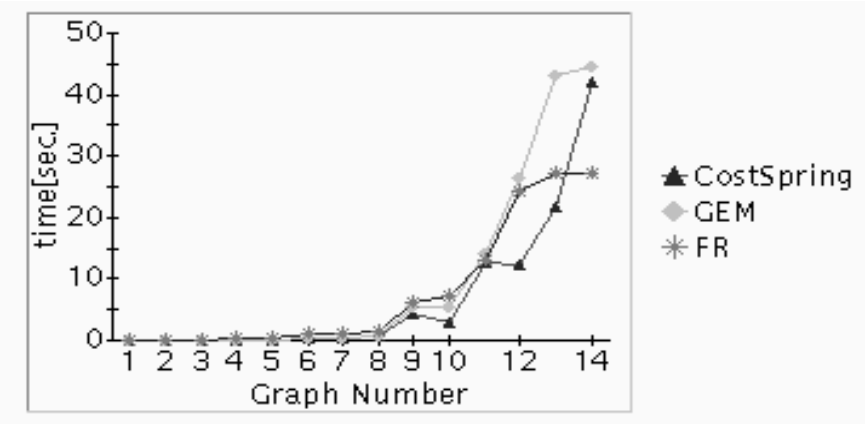

Figure 5.7: Running times of CostSpring, GEM, and FR for sparse graphs.

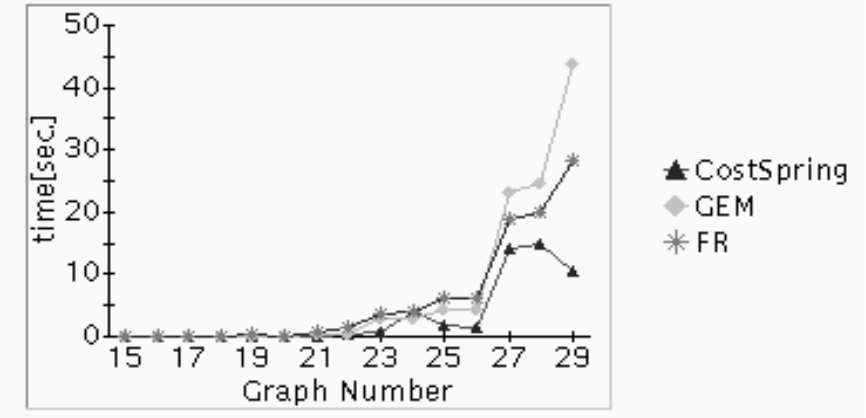

Figure 5.8: Running times of CostSpring, GEM, and FR for normal graphs.

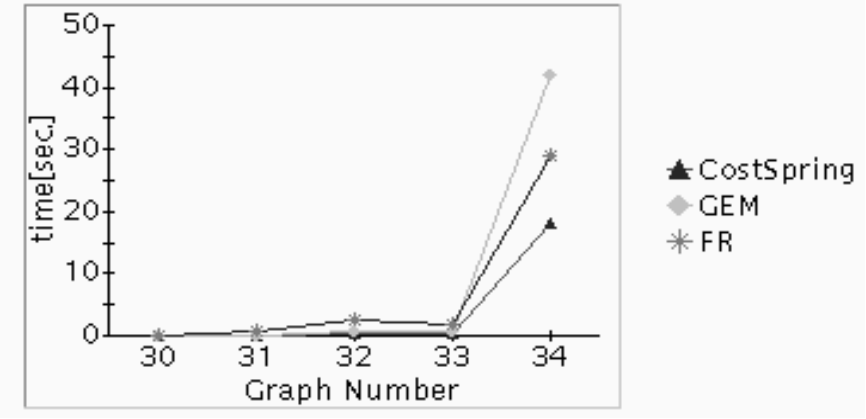

Figure 5.9: Running times of CostSpring, GEM, and FR for dense graphs.

sample resulting layouts of CostSpring algorithm for the graphs of our test suite.

5.4.1 Edge Crossings

Table 5.4 shows the number of edge crossings for the resulting layouts of the graphs of our test suite produced by CostSpring, GEM, and FR algorithms. An asterisk on the side of a graph name indicates that the graph is planar, and can be drawn with no edge crossings. As we can see, for larger graphs GEM and FR produce many edge crossings in the layouts of planar graphs.

5.4.2 Uniformity of Edge Lengths

One of the goals of spring-based graph drawing algorithms is to produce layouts with uniform edge lengths. In this section we compare the uniformity of the edge lengths of the layouts of CostSpring algorithm with those of GEM and FR. We use the ratio of the longest edge over the shortest edge, and the standard deviation of edge lengths to measure the uniformity of edge lengths in a layout. To avoid bias, we compute the normalized deviation of the edge lengths. Figures [5.10-5.12] and [5.13-5.15] show the results of these two measurements. As we can see from these Figures, the three algorithms produce layouts with approximately the same degree of uniformity in edge lengths. However in some cases, GEM's layouts have smaller edge length differences. This is due to the use of a central gravity force as well as a randomized factor that is added to each node position at displacement by GEM [15].

		CostSpring	GEM	FR
#	*Name*	*Crossings*	*Crossings*	*Crossings*
1	Binary Tree 15*	0	0	0
2	Path 16*	0	0.2	0
3	Cycle 16*	0	0	0
4	Star 24*	0	0	0
5	Binary Tree 31*	0	0	0.4
6	Path 48*	0	1	0.4
7	Cycle 48*	0	0.4	0.6
8	Binary Tree 63*	0.2	1	1
9	Binary Tree 127*	1.2	3.2	8
10	Path 128*	0.6	2	7.8
11	Binary Tree 180*	0.6	10.4	27.4
12	Cycle 220*	3.4	8	22.6
13	Path 256*	1	6.6	29.4
14	Binary Tree 255*	0	7.2	29.4
15	Wheel	9.2	5.6	9.2
16	4×4 Square Grid*	0	0	0
17	Hypercube4D	23.4	24	24
18	Dodecahedron	8.4	5.8	8.4
19	Triangular Grid 28*	0	0	0
20	Hypercube5D	175	176	175.6
21	7×7 Square Grid*	0	5.2	0
22	Triangular Grid 55*	0	0	0
23	Hexagonal Grid 96*	0	3.2	0
24	5×20 Square Grid*	5	9.8	12
25	10×12 Square Grid*	0	51.2	1.2
26	Triangular Grid 120*	0	0.8	0
27	Triangular Grid 210*	0	92.4	7.6
28	Hexagonal Grid 216*	0	23.4	30.6
29	16×16 Square Grid*	0	21.8	34.4
30	K12	372.6	403.4	390.6
31	K24	7867	8108	7877
32	K50	16557	17108	16637
33	Hypercube6D	993.6	992.4	996.2
34	Hypercube8D	24950	24964	25109

Table 5.4: Number of crossings in layouts of CostSpring, GEM, and FR. A * on the side of a graph name indicates that the graph is planar.

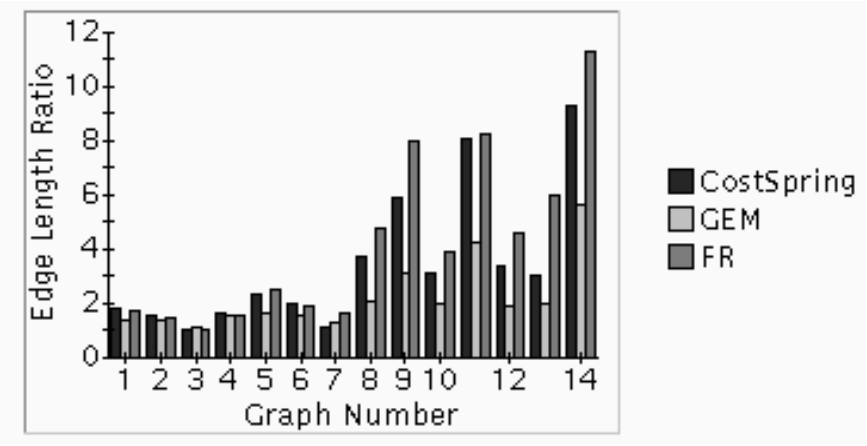

Figure 5.10: Longest edge to shortest edge ratios for sparse graphs.

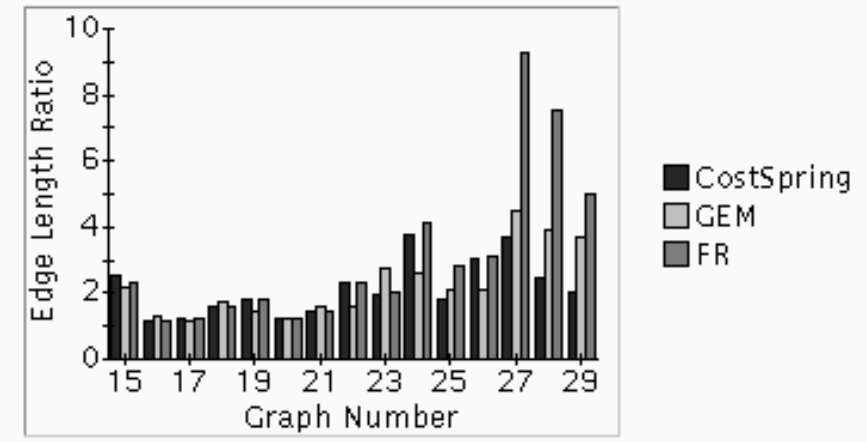

Figure 5.11: Longest edge to shortest edge ratios for normal graphs.

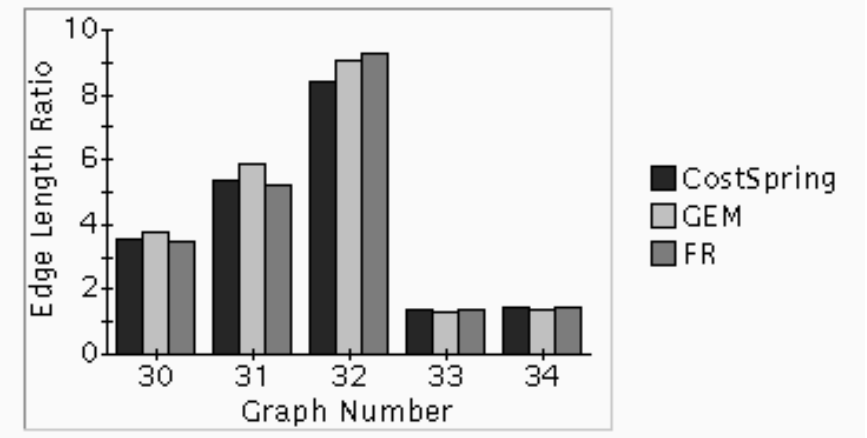

Figure 5.12: Longest edge to shortest edge ratios for dense graphs.

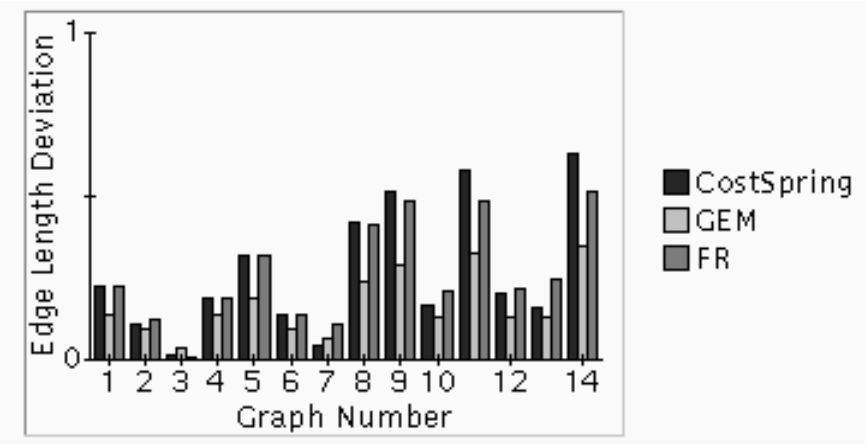

Figure 5.13: A comparison of edge length deviations for sparse graphs.

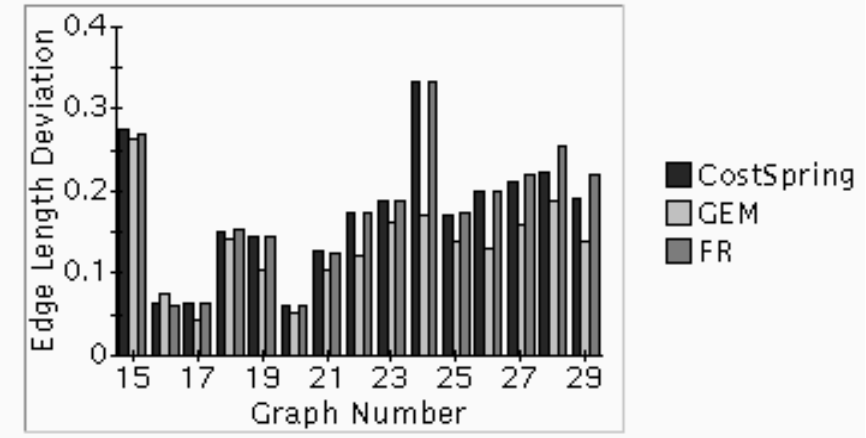

Figure 5.14: A comparison of edge length deviations for normal graphs.

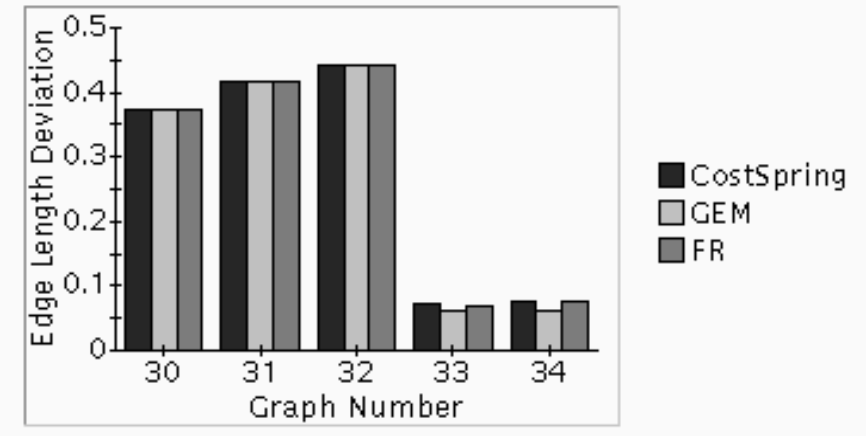

Figure 5.15: A comparison of edge length deviations for dense graphs.

5.4.3 The Cost Value

In this Section, we make a comparison between the layout cost values of the layouts of the 34 graphs of our test suite produced by CostSpring, GEM, and FR. Figures [5.16-5.18] show the result of this experiment. This result indicates that CostSpring's layouts have approximately the same cost values as those of GEM and FR for non-planar graphs as well as the planar graphs for which the three algorithms produce layouts with approximately the same number of edge crossings. The cost values of layouts of planar graphs produced by GEM and FR with many edge crossings is higher than those of CostSpring which have fewer edge crossings.

5.5 Graph Drawing with Eigenvectors

To provide a non-spring based option for graph drawing, we have implemented the eigenvector graph drawing algorithm [29]. In this Section, we illustrate the running-time of this algorithm as well as the number of edge crossings of its resulting layouts for our test suite of graphs. As noted in [29] in some cases, this algorithm produces graph layouts which have overlapping nodes. In such cases, we run the CostSpring algorithm on the graph layouts with overlapping nodes. Table 5.5 show the running-time and the number of edge crossings in the resulting layout of each graph of our test suite. In this Table, the graphs for which eigenvector algorithm generates layouts with overlapping node positions are indicated with an asterisk beside their names.

As we can see in Table 5.5, the eigenvector algorithm generates layouts with few edge crossings, and its running-time is generally lower than that of Spring algorithms

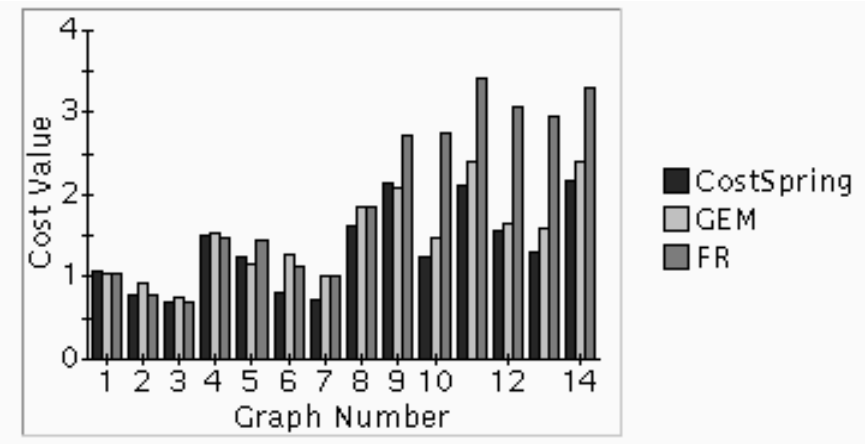

Figure 5.16: A comparison between graph layout cost values for sparse graphs.

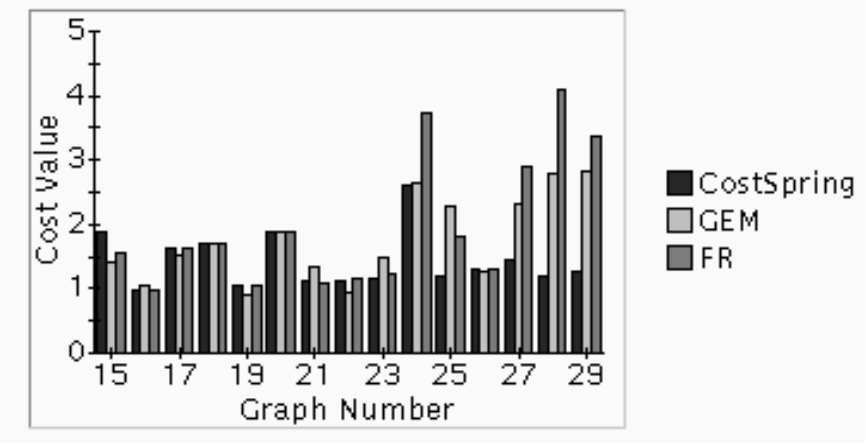

Figure 5.17: A comparison between graph layout cost values for normal graphs.

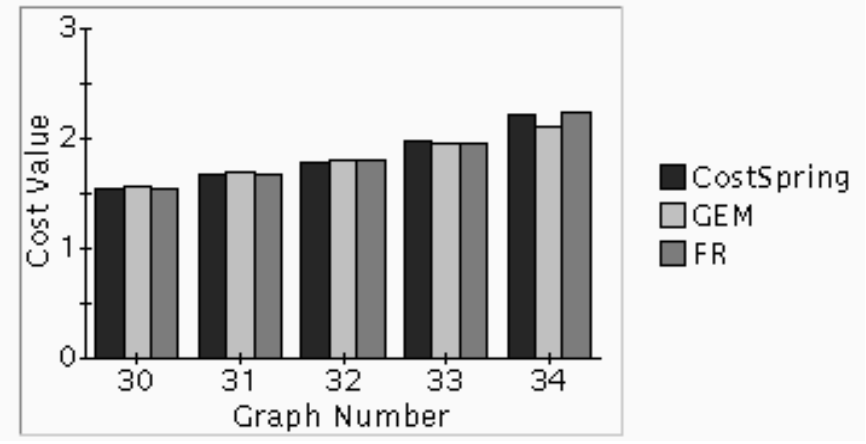

Figure 5.18: A comparison between graph layout cost values for dense graphs.

#	*Name*	*time*[*s*]	*Crossings*
1	Binary Tree 15*	0.0189	0
2	Path 16	0.004	0
3	Cycle 16	0.004	0
4	Star 24*	0.0309	0
5	Binary Tree 31*	0.0729	0
6	Path 48	0.082	0
7	Cycle 48	0.0799	0
8	Binary Tree 63*	0.768	1
9	Binary Tree 127*	6.032	0
10	Path 128	1.2339	0
11	Binary Tree 180*	15.607	2
12	Cycle 220	6.578	0
13	Path 256	11.543	0
14	Binary Tree 255*	56.172	0
15	Wheel	0.002	6
16	4×4 Square Grid	0.003	0
17	Hypercube4D	0.012	22
18	Dodecahedron	0.006	8
19	Triangular Grid 28	0.014	0
20	Hypercube5D	0.016	136
21	7×7 Square Grid	0.068	0
22	Triangular Grid 55	0.1	0
23	Hexagonal Grid 96	0.509	0
24	5×20 Square Grid	0.5509	0
25	10×12 Square Grid	0.967	0
26	Triangular Grid 120	0.9319	0
27	Triangular Grid 210	5.808	0
28	Hexagonal Grid 216	6.9609	0
29	16×16 Square Grid	10.314	0
30	K12	0.002	282
31	K24	0.008	5728
32	K50*	0.303	16775
33	Hypercube6D	0.101	844
34	Hypercube8D	6.9619	21952

Table 5.5: Running time and number of edge crossings of Eigen-Spring algorithm. An asterisk besides a graph name indicates that eigenvector algorithm generated a layout with overlapping nodes for that graph.

demonstrated in Table 5.3. The major drawback of this approach is the overlapping nodes in some of its resulting graph layouts.

5.6 Finding the Layout That Is Not in an Unwanted Local Minimum

As mentioned in Section 3.3, the layouts of some graphs such as narrow grids and binary trees produced by Spring algorithms have a tendency of ending up in some unpleasing local minima. Our *CostChosen* initial graph layout algorithm, described in Section 3.3, is to reduce the chance of finding graphs layouts in unwanted local minima.

To experiment with CostChosen algorithm, we test this algorithm on three graphs whose layouts produced by Spring algorithms have a high tendency of ending up in unpleasing local minima: 3×10 square grid, 3×20 square grid, and complete binary tree with 127 nodes. First, we ran CostSpring algorithm on 5 different random initial graph layouts of each of these graphs. The ratio of the number of resulting layouts of 3×10 grid which were not in unpleasing local minima was 2/5. This number was 1/5 and 2/5 for 3×20 grid and binary tree with 127 nodes respectively. To show the effectiveness of the CostChosen algorithm, we tested this algorithm on the above three graphs with different number of initial random graph layouts and iteration counts. In each case, we found the initial layout produced by CostChosen algorithm, ran the CostSpring algorithm on this initial layout, and recorded whether the resulting layout was in an unwanted local minimum or not. For each combination of initial

3×10 *Grid*	Number of Initial Layouts					
	2		5		10	
Iterations	Success	t[sec]	Success	t[sec]	Success	t[sec]
$\frac{1}{2}\|V\|$	4/5	0.1349	5/5	0.217	5/5	0.697

Table 5.6: Success ratio and running-time of CostChosen algorithm on a 3 by 10 square grid.

3×20 *Grid*	Number of Initial Layouts							
	2		5		10		15	
Iterations	Success	t[sec]	Success	t[sec]	Success	t[sec]	Success	t[sec]
$\frac{1}{2}\|V\|$	1/5	0.553	2/5	1.085	3/5	2.37	4/5	3.459
$\|V\|$	0/5	0.572	4/5	1.51	3/5	2.92	5/5	4.43

Table 5.7: Success ratio and running-time of CostChosen algorithm on a 3 by 20 square grid.

random layout count and iteration count, we ran the CostChosen algorithm 5 times and reported the ratio at which CostChosen found initial layouts that did not end up in unpleasing local minima. We show the running-time of CostChosen algorithm for each test run. Tables 5.6 ,5.7, and 5.8 show the results of our experiment.

As we can see in these Tables, with the right number of iteration counts and initial random layouts, in all three cases CostChosen eliminated any unpleasing local

Complete Binary Tree	Number of Initial Layouts					
	2		5		10	
Iterations	Success	t[sec]	Success	t[sec]	Success	t[sec]
$\frac{1}{2}\|V\|$	1/5	3.642	3/5	9.048	3/5	16.179
$\|V\|$	4/5	6.057	4/5	16.694	5/5	33.444

Table 5.8: Success ratio and running-time of CostChosen algorithm on a complete binary tree with 127 nodes.

minimum in the layouts of these three graphs in all of its 5 runs. A problem that we present as a future work to this thesis is that of automatic determination of what we call the *"right number of iteration counts and initial random layout"* in the CostChosen algorithm for a given graph. Given the adjacency information of a graph we would like to have a method which can determine whether the CostChosen initial layout is needed for that graph, and if so, what are the suitable number of iteration counts and number of initial layouts in CostChosen algorithm for that graph. Through our experiments we have discovered that:

- Dense [3] graphs don't require the use of CostChosen initial layout.
- Sparse graphs such as binary trees often end up in local minima, and are candidates for CostChosen initial layout.
- Among the graphs with normal densities which we tested, narrow grids such as 3×10 and 3×20 grids show that they can benefit from the CostChosen initial graph layout algorithm.

5.7 Summary

In this Chapter we presented the results of a number of experiments that evaluated the running-time and the graph layout quality of CostSpring algorithm, the performance of eigenvector graph drawing algorithm, and finally the effectiveness of the CostChosen initial graph layout algorithm.

[3] see Section 3.3 for our definition of dense, normal, and sparse graphs.

CostSpring showed that it converges faster than GEM and FR, and its stopping method is effective with the exception of graphs for which oscillation occurs. Starting from initial random graph layouts, CostSpring generates layouts with few or no edge crossings (in case of planar graphs). The number of edge crossings in layouts of planar graphs produced by CostSpring is noticeably smaller than those generated by GEM and FR. The edge length uniformity of the layouts of this algorithm is very similar to layouts of FR algorithm. Graph layouts produced by GEM show a slightly better edge length uniformity than those of CostSpring and FR due to GEM's use of a central gravity force as well as a randomized factor that is added to each node position at displacement.

Eigenvector graph drawing algorithm is faster that Spring algorithms, however, its major drawback is the overlapping nodes that occur in layouts of some graphs generated by this algorithm.

Finally, our experiments with CostChosen initial graph layout algorithm show that this method can reduce or eliminate the chance of finding a graph layout that is in an unpleasing local minimum. CostChosen's degree of success for a given graph is dependent on the number of initial random layouts as well as the iteration counts used. A method that automatically finds the right number of initial random layouts and iteration counts of CostChosen algorithm is currently an open problem.

Chapter 6

Conclusion and Future Work

6.1 Contributions

In this thesis, we introduce a node layout and a graph layout cost function and use these functions to improve on the running-time and the quality of layouts of Spring algorithms. Specifically, we use the differences between the cost values of the individual nodes over the successive iterations of our Spring algorithm, *CostSpring*, to determine the local temperature of individual nodes. We divide our algorithm into two phases. In the first phase we allow for large node displacement, and the second phase is to fine-tune the layout. To terminate the algorithm, we use the graph layout cost values to determine the convergence of the algorithm. Furthermore, the graph layout cost values are used to select the layout with best quality among a number of layouts of a given graph produced by our Spring algorithm starting from different initial random layouts. The layout found by this method can be fed into any Spring algorithm as its initial layout. This approach which we call *CostChosen* initial layout aims at reducing the chance of finding layouts in undesirable local minima.

In addition, we developed the LayoutShow software. In this software, we implement our improved Spring algorithm as well as our cost chosen initial layout algorithm. A variety of other Spring algorithms, initial layout algorithms, and the eigenvector algorithm [29] are also implemented in LayoutShow.

6.2 Conclusion

In conclusion, we recommend the CostSpring algorithm for embedding graphs with up to 256 nodes. Our experiments show that our Spring algorithm generates layouts with considerably fewer edge crossings for planar graphs and shorter execution times than GEM [15] and FR [16] Spring algorithms. In addition, CostChosen initial layout is an effective method where time is not a primary factor. This method can eliminate or significantly reduce the chance of finding layouts in unpleasing local minima, however, it is computationally intensive.

In addition, the LayoutShow is a flexible software environment for generating graphs using Spring algorithms, eigenvector algorithm, and for experimentation with such algorithms. The applet version of this software allows for local file I/O. Therefore, if the user's data has been generated by another application in the GML [19] file format, the user can use the LayoutShow applet to layout his/her graph and save the result on the local disk. This eliminates the need for down-loading the application version of LayoutShow.

6.3 Future Work

The following is a list of avenues for future work on this thesis [1]:

- Detecting oscillation in CostSpring algorithm. This can improve accuracy of our stopping method for this algorithm for graphs such as binary trees with more than 100 nodes.
- Adding the efficient computation of repulsive forces from [37] to CostSpring algorithm to further improve its running-time.
- Computing an accurate cost value of the graph layout *efficiently* after each iteration of CostSpring algorithm. This may be beneficial in further improving the accuracy of our method of stopping the algorithm. However, before attempting this modification one must justify whether the accurate values of graph layout costs would in fact result in any improvement.
- Adding GEM's [15] gravity force and randomized factor to CostSpring for obtaining better edge length uniformity.
- Finding an algorithm to automatically determine whether a graph can benefit from CostChosen initial layout algorithm, and if so, finding the number of initial layouts and the iteration counts needed by CostChosen algorithm for the given graph.
- Extending LayoutShow to support graph editing, better node labeling, and layout scaling.

[1] An extension to this work can be found in [6].

Appendix A

Examples of Our Node and Layout Cost Values

Figure A.1 demonstrates five examples of the maximum and minimum node cost values, Q_v values of Equation 3.10, in the layouts of five different graphs. The layouts have been generated in Matlab [1] using the Graphlet [20] implementation of FR [16] spring algorithm translated from C++ to Matlab script. The default configurations of Graphlet version of FR are used except for the number of iterations that is fixed at 500.

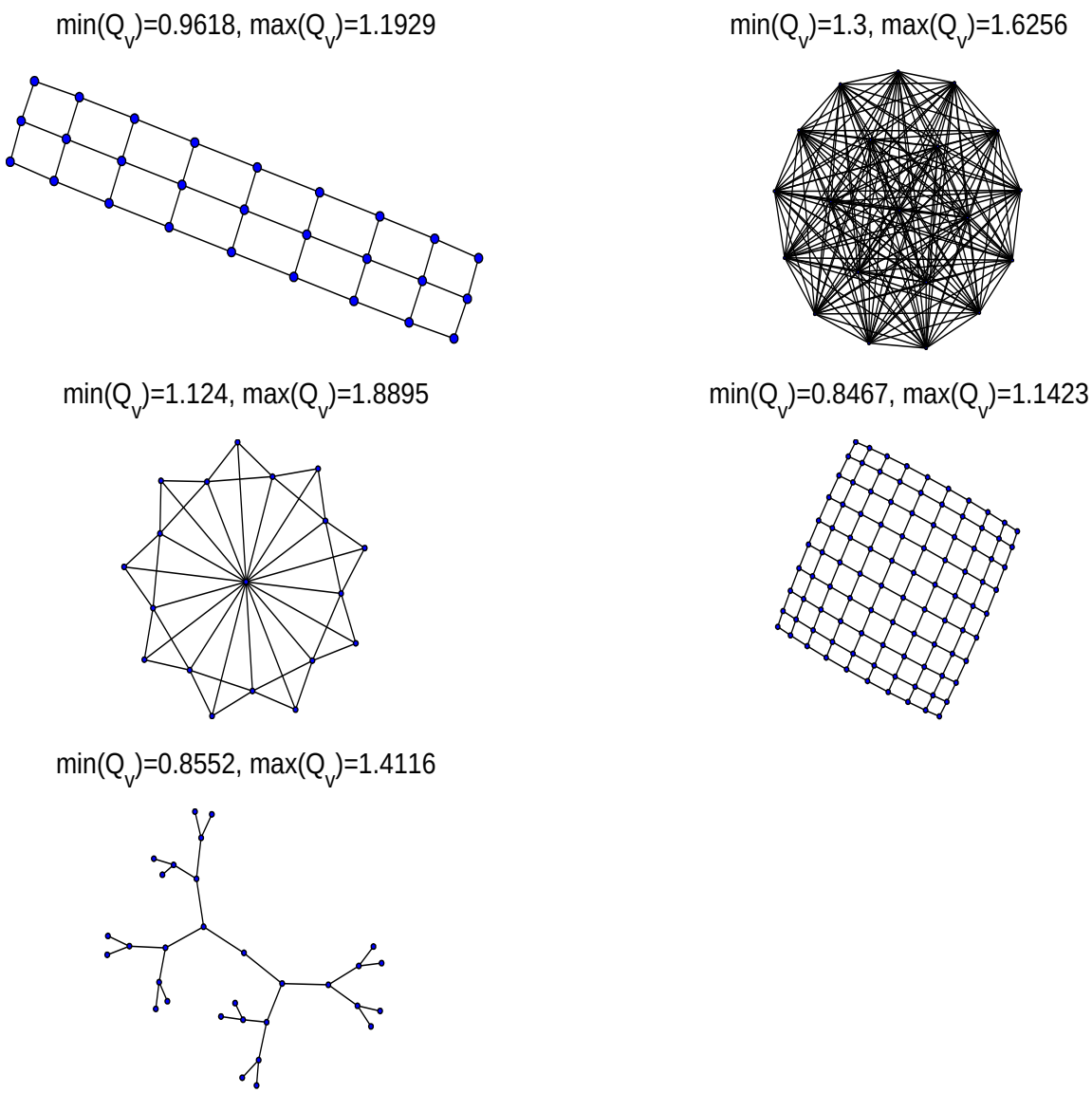

Figure A.1: Five Layouts with titles indicating their maximum and minimum Q_v values of Equation 3.10 among all the graph vertices.

Appendix B

Performance of GEM Compared to FR for Large Graphs

Our experiments show that FR algorithm [16] can outperform GEM algorithm [15] as the graph sizes grow larger. This result contradicts with what was presented in [15] where GEM was found to be faster than FR for graphs up to 255 nodes. We have used the Graphlet [20] versions of these two algorithms which are the same implementations and configurations used in the experiments of [15] (see Section 5.2.2 for these configurations). In addition, we used the same machine as the one used for the experiments of [15]: a SparcStation 10. Table B.1 shows the result of our experiment. The graph numbers in this Table correspond to the ones in Table 5.1. These graphs are the ones used in the experiments of [15].

Even though, each iteration of GEM and FR take $O(n^2)$ time, GEM has a higher overhead (larger multiplication constant factor). In this experiment the number of iterations for FR is fixed at 1000 while the number of iterations for GEM is 3 times its number of nodes. Therefore, for small graphs the number of iterations of FR is

		GEM	FR
#	*Name*	*time*[*s*]	*time*[*s*]
1	Binary Tree 16	0.139	0.422
2	Path 16	0.159	0.455
3	Cycle 16	0.164	0.464
4	Star 24	0.358	0.826
5	Binary Tree 31	0.649	1.23
6	Path 48	1.931	2.553
7	Cycle 48	3.112	2.535
8	Binary Tree 63	3.976	4.11
9	Binary Tree 127	28.272	14.677
10	Path 128	28.957	14.561
13	Path 256	229.066	64.491
14	Binary Tree 255	217.401	61.827
15	Wheel	0.169	0.467
16	4×4 Square Grid	0.187	0.614
17	Hypercube4D	0.245	0.561
18	Dodecahedron	0.275	0.699
19	Triangular Grid 28	0.618	1.283
20	Hypercube5D	0.882	1.663
21	7×7 Square Grid	2.183	2.882
22	Triangular Grid 55	0.316	0.3916
23	Hexagonal Grid 96	12.788	9.579
26	Triangular Grid 120	25.568	15.511
27	Triangular Grid 210	134.359	46.596
29	16×16 Square Grid	236.05	64.007
30	K12	0.254	0.695
31	K24	1.296	2.874
33	Hypercube6D	4.909	5.166

Table B.1: Running times of GEM, and FR.

so much larger than that of GEM to make its total running time slower than GEM. However, as the graphs grow larger and the number of iterations of GEM becomes greater, this algorithm becomes slower than FR's 1000 iterations for the same graph.

Appendix C

Sample Layouts Generated by CostSpring

In this Appendix we present the resulting layouts of CostSpring algorithm for the graphs of our test suite. Table 5.1 lists these graphs. The initial layouts to the algorithm are the same random initial layouts used in the experiments of Chapter 5. The software and hardware configurations used are those mentioned in Section 5.2.

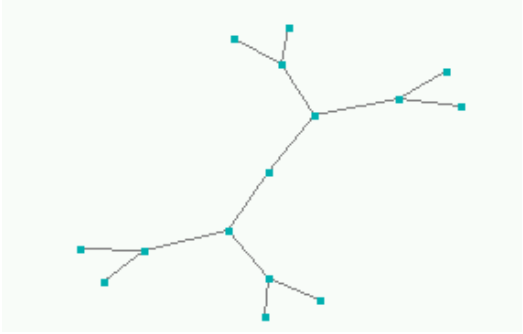

Figure C.1: Binary Tree $|V| = 15$ $|E| = 14$

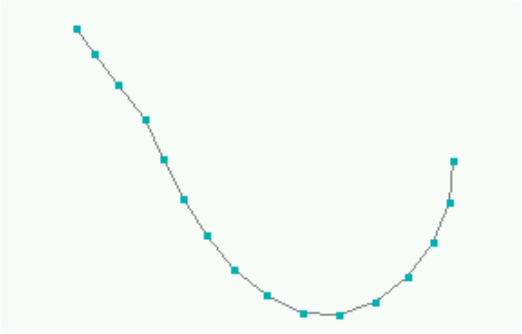

Figure C.2: Path $|V| = 16$ $|E| = 15$

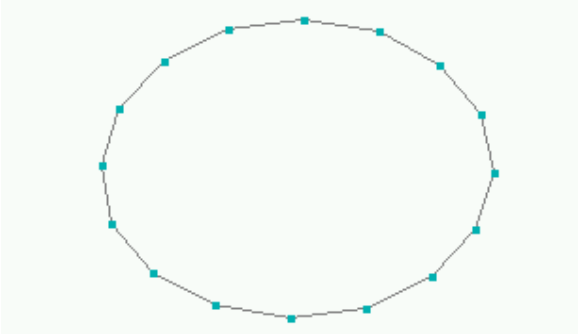

Figure C.3: Cycle $|V| = 16$ $|E| = 16$

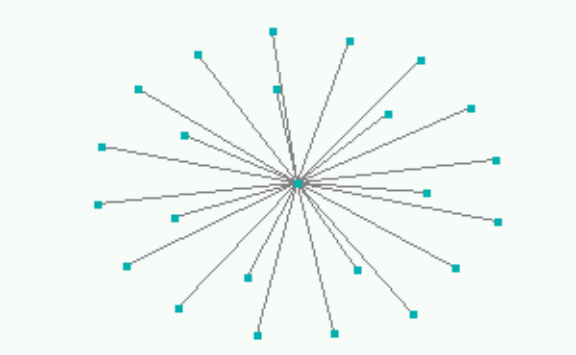

Figure C.4: Star $|V| = 24$ $|E| = 23$

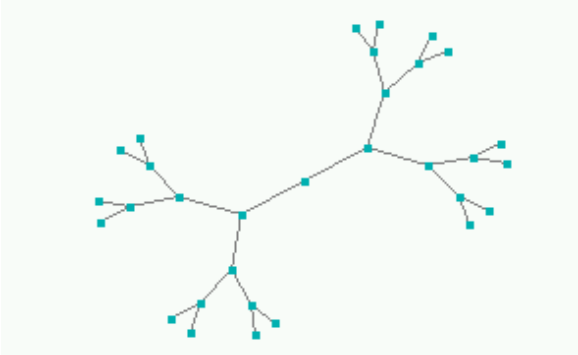

Figure C.5: Binary Tree $|V| = 31$ $|E| = 30$

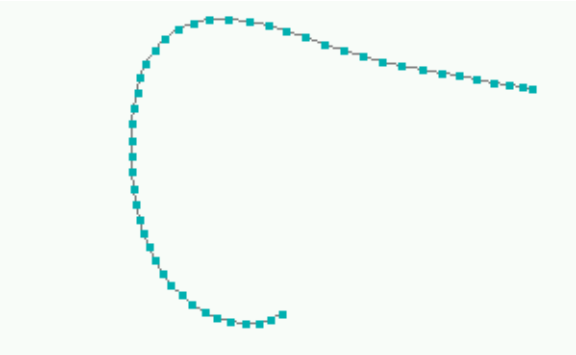

Figure C.6: Path $|V| = 48$ $|E| = 47$

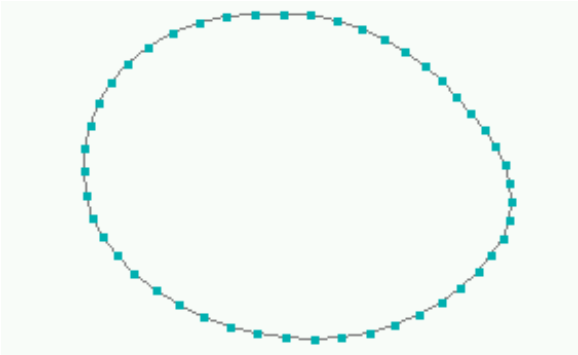

Figure C.7: Cycle $|V| = 48$ $|E| = 48$

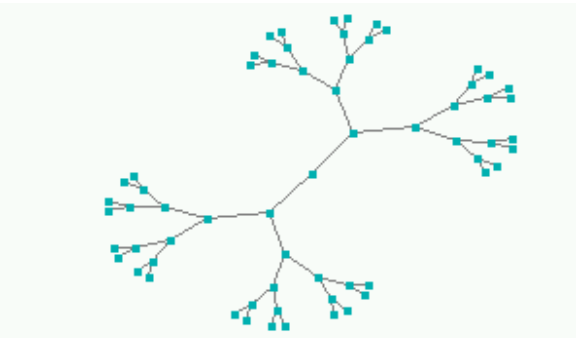

Figure C.8: Binary Tree $|V| = 63$ $|E| = 62$

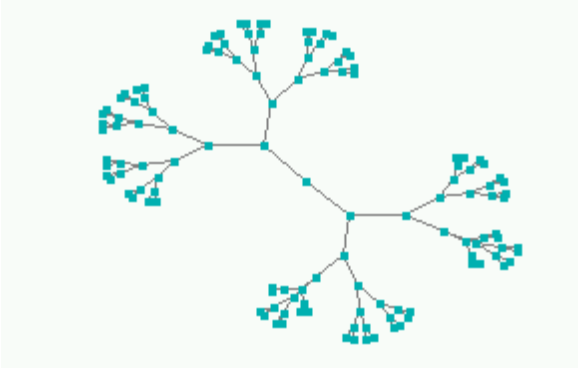

Figure C.9: Binary Tree $|V| = 127$ $|E| = 126$

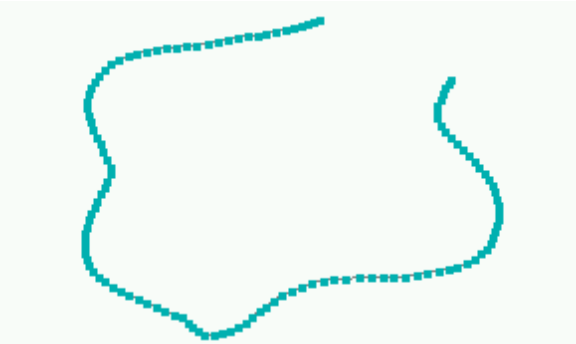

Figure C.10: Path $|V| = 128$ $|E| = 127$

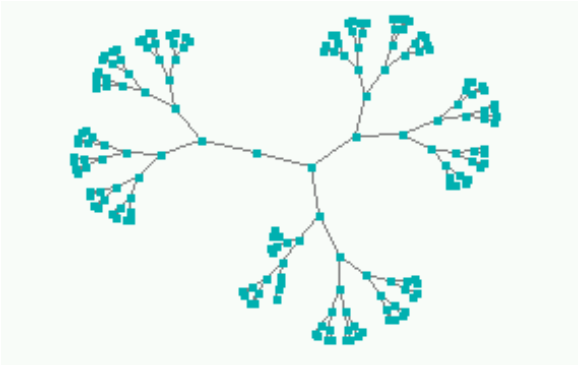

Figure C.11: Binary Tree $|V| = 180$ $|E| = 179$

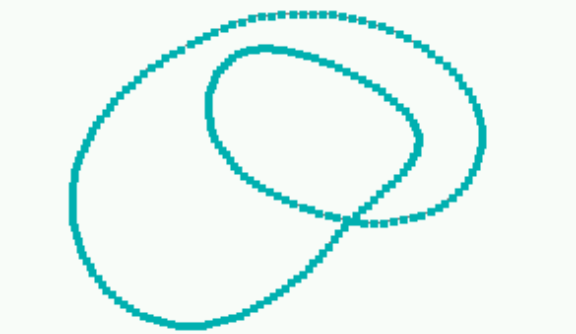

Figure C.12: Cycle $|V| = 220$ $|E| = 220$

Figure C.13: Path $|V| = 256$ $|E| = 255$

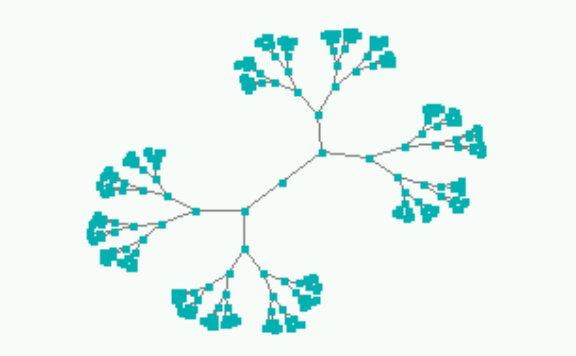

Figure C.14: Binary Tree $|V| = 255$ $|E| = 254$

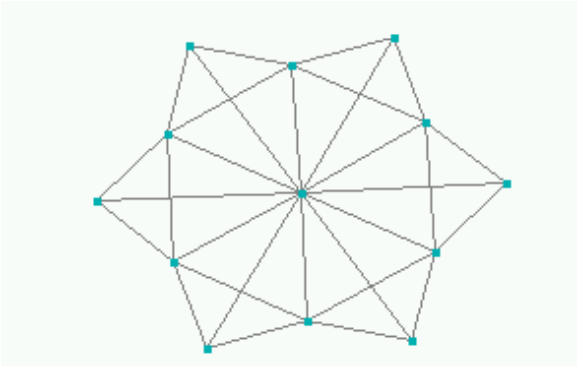

Figure C.15: Wheel $|V| = 13$ $|E| = 24$

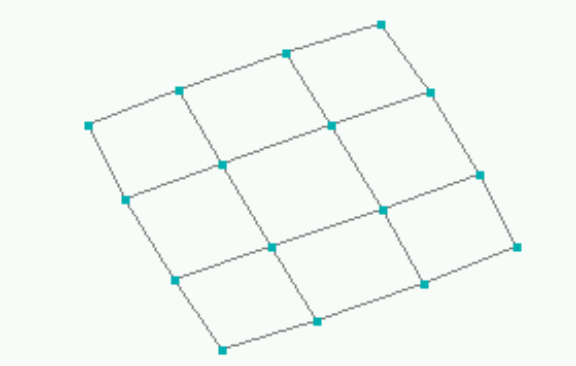

Figure C.16: Square Grid $|V| = 16$ $|E| = 24$

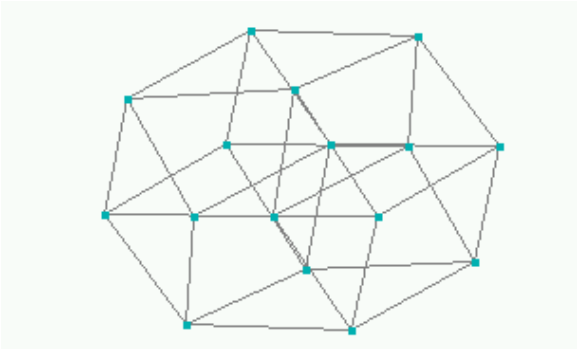

Figure C.17: Hypercube4D $|V| = 16$ $|E| = 32$

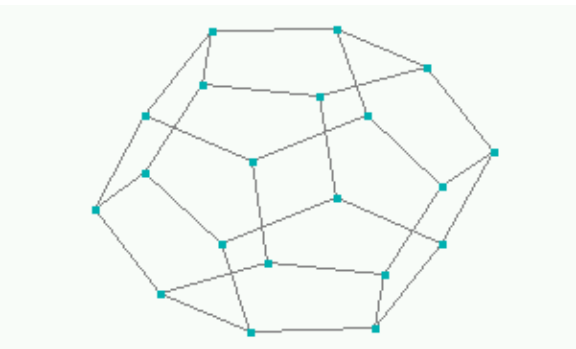

Figure C.18: Dodecahedron $|V| = 20$ $|E| = 30$

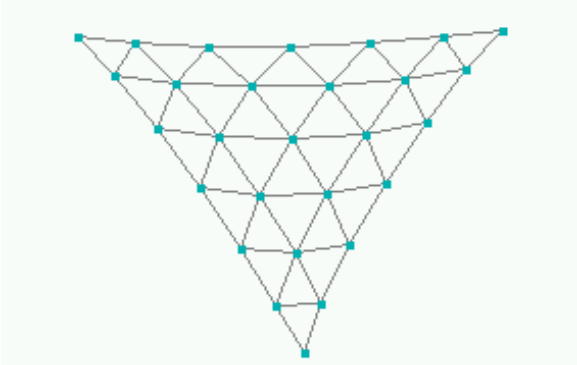

Figure C.19: Triangular Grid $|V| = 28$ $|E| = 63$

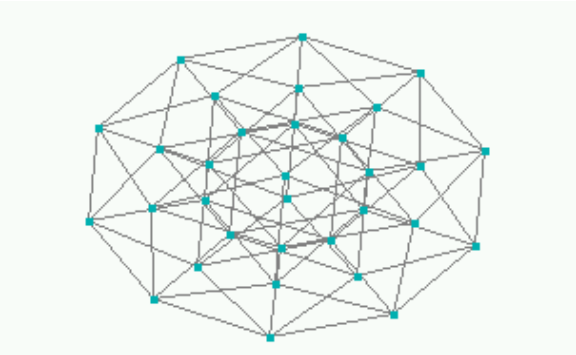

Figure C.20: Hypercube5D $|V| = 32$ $|E| = 80$

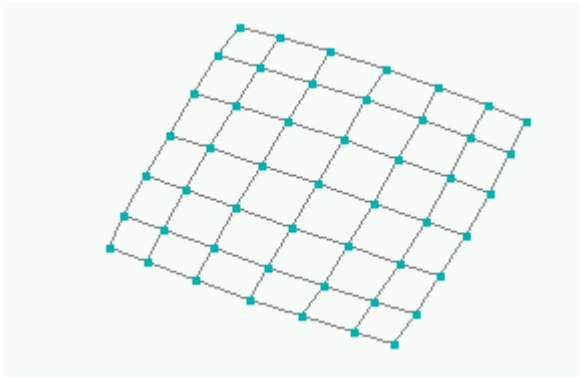

Figure C.21: Square Grid $|V| = 49$ $|E| = 84$

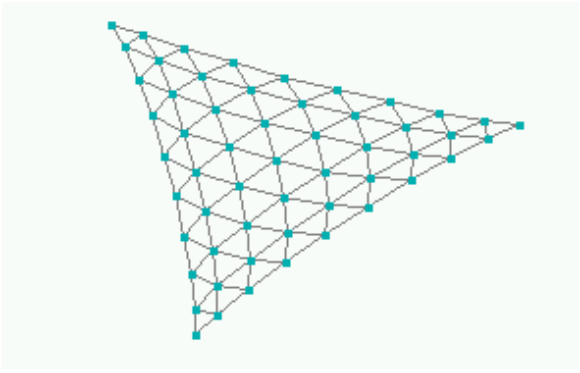

Figure C.22: TriangularGrid $|V| = 55$ $|E| = 135$

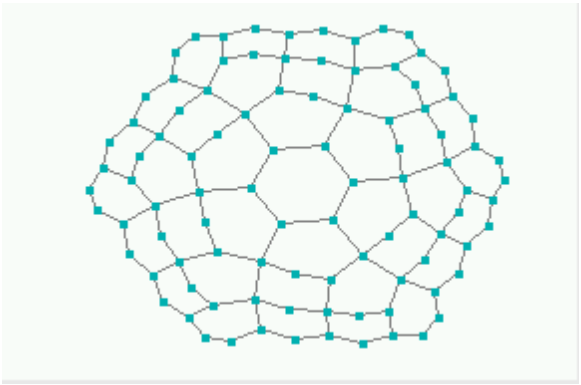

Figure C.23: Hexagonal Grid $|V| = 96$ $|E| = 132$

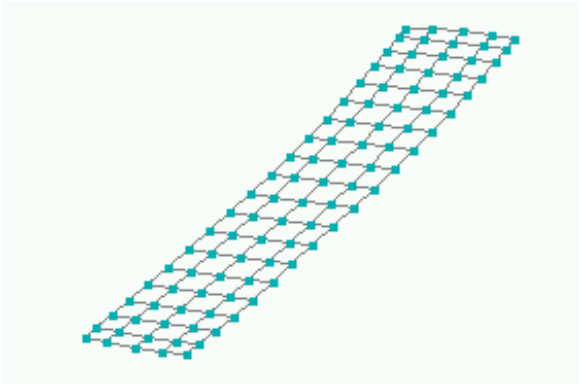

Figure C.24: Square Grid $|V| = 100$ $|E| = 175$

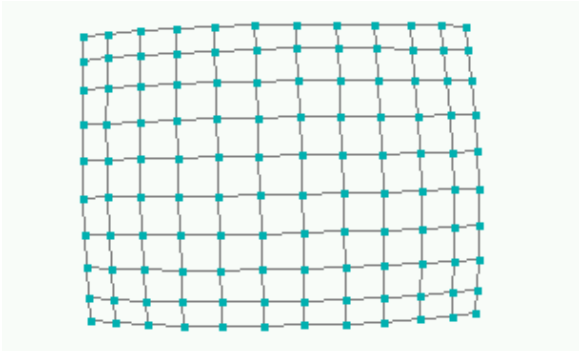

Figure C.25: Square Grid $|V| = 120$ $|E| = 218$

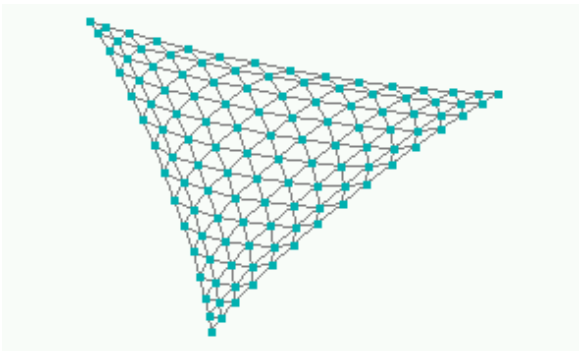

Figure C.26: Triangular Grid $|V| = 120$ $|E| = 315$

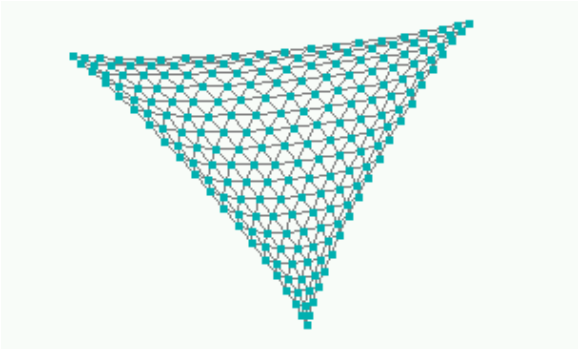

Figure C.27: Triangular Grid $|V| = 210$ $|E| = 570$

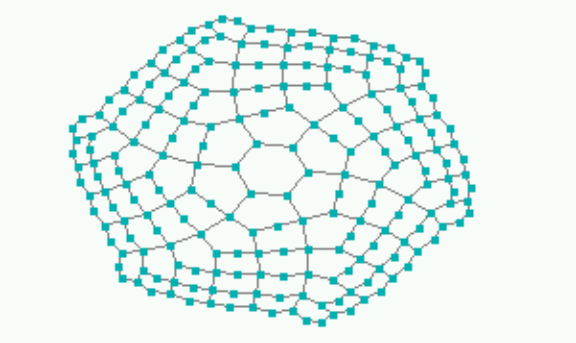

Figure C.28: Hexagonal Grid $|V| = 216$ $|E| = 306$

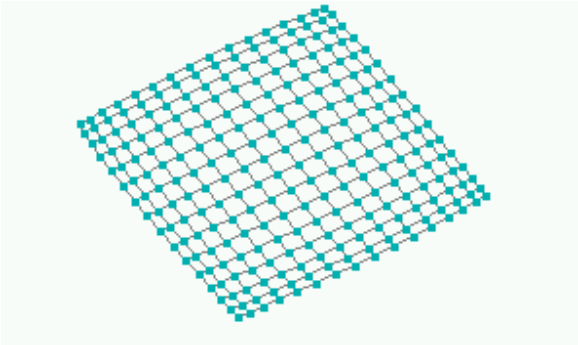

Figure C.29: Square Grid $|V| = 256$ $|E| = 480$

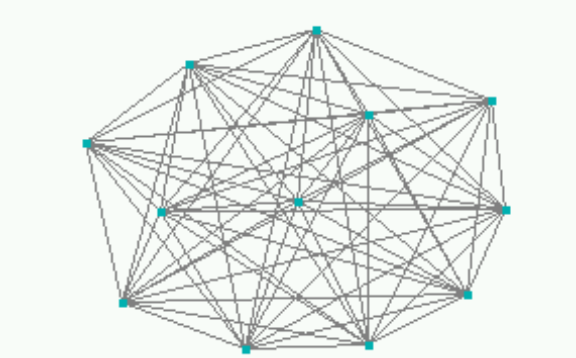

Figure C.30: Complete Graph $|V| = 12$ $|E| = 66$

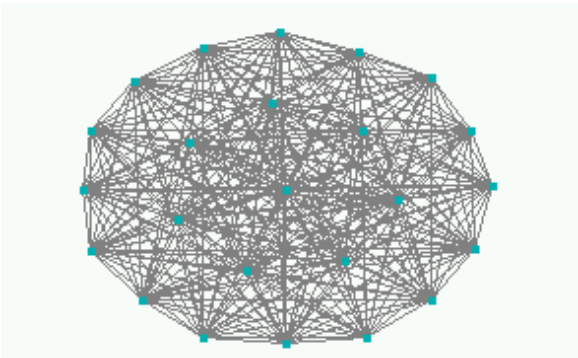

Figure C.31: Complete Graph $|V| = 24$ $|E| = 276$

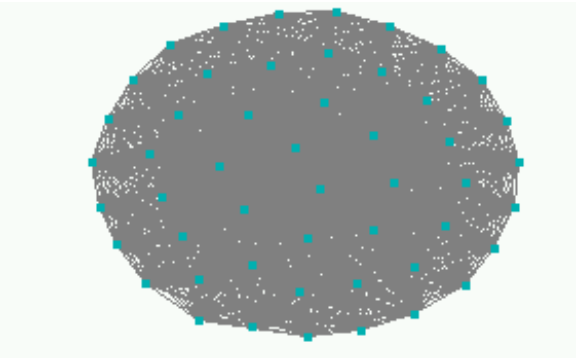

Figure C.32: Complete Graph $|V| = 50$ $|E| = 1225$

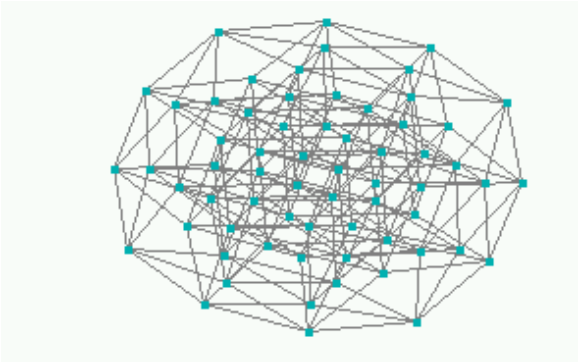

Figure C.33: Hypercube6D $|V| = 64$ $|E| = 192$

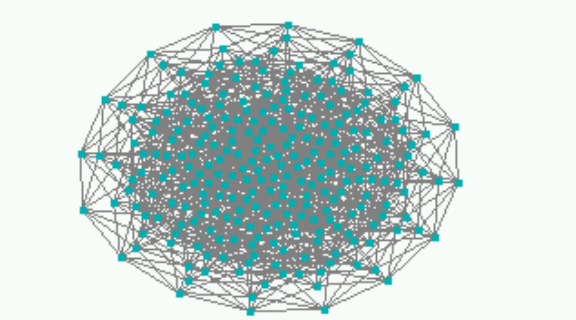

Figure C.34: Hypercube8D $|V| = 256$ $|E| = 1024$

Appendix D

Default Values of Configurable Parameters of CostSpring, GEM, and FR

In this Appendix we present the default values of the configurable parameters of CostSpring, GEM, and FR. These default values are used for the experiments of Chapter 5. We must note that adjusting these parameters for any of the algorithms will change its running-time and/or resulting layout.

CostSpring

$C = 0.3$, $CostOptimalDistance = 64$, $C1 = 15$, $beta = 1$, $C2 = 20$, $C3 = 15$, $gamma = 0.1, C4 = 2, C5 = 1, alpha = 0.5$, $MaxNumIterToCheck = 10, tol1 = 0.1$ for sparse graphs, $tol1 = 0.5$ for normal graphs, $tol1 = 3$ for dense graphs, $tol2 = 0.05$ for sparse graphs, $tol2 = 0.01$ for normal and dense graphs, $MaxIterationsPhase1 = 3 * |V|$, $MaxIterationsPhase2 = 5 * |V|$

GEM

$i-maxTemp = 1.0$, $a-maxTemp = 1.5$, $o-maxTemp = 1.5$, $i-startTemp = 0.3$, $a-startTemp = 1.0$, $o-startTemp = 1.0$, $i-finalTemp = 0.1$, $a-finalTemp = 0.02$, $o-finalTemp = 0.0$, $i-maxIter = 10$, $a-maxIter = 3$, $o-maxIter = 3$, $i-gravity = 0.1$, $i-oscillation = 0.4$, $i-rotation = 0.5$, $i-shake = 0.2$, $a-gravity = 0.1$, $a-oscillation = 0.4$, $a-rotation = 0.9$, $a-shake = 0.3$, $o-gravity = 0.1$, $o-oscillation = 0.4$, $o-rotation = 0.9$, $o-shake = 0.3$, $GEMOptimalDistance = 64$, $GEMMaxIterationFactor = 3$,

FR

$FROptimalDistance = 64$, $FRStopForce = 3.00$, $FRVibration = 0.010$, $FRMax-Iteration = 1000$

Bibliography

[1] *MATLAB: Reference Guide.* The MathWorks, Massachusetts, 1992.

[2] J. Barnes and P. Hut. A hierarchial $O(n \log n)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.

[3] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: An annotated bibliography. *Computational Geometry: Theory and Applications*, 4:235–282, 1994.

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

[5] L. Behzadi. LayoutShow: a signed applet/application for graph drawing and experimentation. Technical report, York University, 1999. Submitted to Graph Drawing '99.

[6] L. Behzadi and J. W.H. Liu. On some improvements of the spring-based graph layout algorithm. Technical report, York University, 1999. Submitted to Graph Drawing '99.

[7] F.J. Brandenburg, M. Himsolt, and C. Rohrer. An experimental comparison of force-directed and randomized graph drawing algorithms. In *Proceedings of Graph Drawing '95*, pages 76–87. Springer-Verlag, 1996.

[8] S. Bridgeman, A. Garg, and R. Tamassia. A graph drawing and translation service on the WWW. In *Proceedings of Graph Drawing '96*, pages 45–52. Springer-Verlag, 1997.

[9] M. Campione and K. Walrath. *The Java Tutorial : Object-Oriented Programming for the Internet.* Addison-Wesley, Massachusetts, March 1998.

[10] M. Campione, K. Walrath, and A. Huml. *The Java Tutorial Continued : The Rest of the Jdk.* Addison-Wesley, Massachusetts, December 1998.

[11] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction To Algorithms.* The MIT Press, Cambridge, Massachusetts, 1992.

[12] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics*, 15:301–331, 1996.

[13] U. Dogrusoz, M. Doorley, Q. Feng, A. Frick, B. Madden, and G. Sander. Toolkit for development of software diagramming applications. In *Proceedings of International Conference on Software Engineering & Knowledge Engineering.* Knowledge Systems Institute, 1998.

[14] P. Eades. A heuristic for graph drawing. *Congr. Nuumer.*, 42:149–160, 1984.

[15] A. Frick, A. Ludwig, and H. Mehldau. A fast adaptive layout algorithm for undirected graphs. In *Proceedings of Graph Drawing '94*, pages 388–403. Springer-Verlag, 1995.

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

[17] P. Gill, W. Murray, and M. Wright. *Practical Optimization*. Academic Press, London, 1981.

[18] Mark Hale. *JSci - A science API for Java*. University of Durham, Department of Mathematical Sciences, 1998. Currently available at: `http://fourier.dur.ac.uk:8000/~dma3mjh/jsci`.

[19] M. Himsolt. GML: A portable graph file format. Technical report, University of Passau, 94030 Passau, Germany, 1997. Currently available at: `http://www.fmi.uni-passau.de/Graphlet/GML/gml-tr.html`.

[20] M. Himsolt. The Graphlet system. In *Proceedings of Graph Drawing '96*, pages 233–240. Springer-Verlag, 1997.

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

[22] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulate annealing. *Science*, 220:671–680, 1983.

[23] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, Massachusetts, 1997.

[24] C. McCreary and L. Barowski. VGJ: Visualizing graphs through java. In *Proceedings of Graph Drawing '98*, pages 454–455. Springer-Verlag, 1999.

[25] C. McCreary, R. Chapman, and F. Shieh. Using graph parsing for automatic graph drawing. *IEEE Trans. on System, Man and Cybernetics*, September 1998.

[26] A. Moffat and T. Takaoka. An all pairs shortest path algorithm with expected running time $O(n^2 \log n)$. *Proc. Conf. Found. Comp. Sci.*, pages 101–105, 1985.

[27] Object Space. *JGL 3.1 User Guide*, 1998. Currently available at: `http://www.objectspace.com/developers/jgl/white/doc/user/UserGuide.html`.

[28] R. Otten and L. Van Ginneken. *The Annealing Algorithm*. Kluwer Academic Publishers, Boston, 1989.

[29] T. Pisanski and J. Shawe-Taylor. Characterizing graph drawing with eigenvectors. Technical report, Royal Holloway, University of London, 1998. Currently available at: `http://www.ijp.si/tomo/papers/papers.htm`.

[30] H. Purchase. Which aesthetic has the greatest effect on human understanding? In *Proceedings of Graph Drawing '97*, pages 248–261. Springer-Verlag, 1998.

[31] P.M. Spira. A new algorithm for finding all shortest paths in a graph of positive arcs in average time $O(n^2 \log^2 n)$. *SIAM J. Comput.*, 2:28–32, 1973.

[32] K. Sugiyama and S. Misue. A simple and unified method for drawing graphs: Magnetic-spring algorithm. In *Proceedings of Graph Drawing '94*, pages 364–375. Springer-Verlag, 1995.

[33] Sun Microsystems. *Java Plug-in Documentation*, 1998. Currently available at: `http://java.sun.com/products/plugin/1.1.2/docs/index.html`.

[34] Sun Microsystems. *JDK 1.1 Documentation*, 1998. Currently available at: `http://java.sun.com/docs/index.html`.

[35] Sun Microsystems. *Java Plug-in 1.2 Documentation*, 1999. Currently available at: `http://java.sun.com/products/plugin/1.2/docs/index.docs.html`.

[36] D. Tunkelang. A practical approach to drawing undirected graphs. Technical report, Carnegie Mellon University, School of Computer Science, 1994.

[37] D. Tunkelang. JIGGLE: Java interactive graph layout environment. In *Proceedings of Graph Drawing '98*, pages 413–422. Springer-Verlag, 1999.

[38] W.T. Tutte. Convex representations of graphs. In *Proceedings London Mathematical Society*, volume 10, pages 304–320, 1960.

[39] W.T. Tutte. How to draw a graph. In *Proceedings London Mathematical Society*, volume 13, pages 743–768, 1963.

[40] J.Q. Walker. A node-positioning algorithm for general trees. *Software Practice and Experience*, 20(7):685–705, 1990.

[41] M. A. Weiss. *Data Structures and Algorithm Analysis*. The Benjamin/Cummings Publishing Company, California, 1995.

[42] J. Zukowski. *Java AWT Reference*. O'Reilly and Associates, California, 1997.