A DYNAMIC LAYOUT ALGORITHM FOR GRAPH DRAWING IN THREE DIMENSIONS

Yuejing Meng

A MAJOR REPORT

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

July 2002

0-612-72941-9

ABSTRACT

A Dynamic Layout Algorithm For Graph Drawing in Three Dimensions

Yuejing MENG

Graph drawing is the problem of representing graphs visually. How to efficiently represent graphs for visualization and intuition, as well as the pure beauty of the interplay between graph theory and geometry, has been investigated by mathematicians for centuries. Methodologies for creating graph displays have typically focused on drawing the graph on a two-dimensional surface. Today, interest in computer-based visualization has increased attention on methodologies for the display of graphs in three dimensions.

A dynamic layout algorithm designed by Szirmay-Kalos takes a description of a graph $G = (V, E)$ and assigns coordinates to the vertices and edges so that the graph can be drawn by a graphics program. The algorithm is based on physical simulation of an analogous mechanical system in which the vertices correspond to particles and the presence or absence of edges correspond to driving forces among these vertices. From the initial configuration of the vertices, the system is replacing them in such a way so that the local forces exerted on a vertex for all vertices are at minimum, which is defined as stable state of the system. Allowing the system to reach the stable state in three dimensions would enable the use of interactive computer visualization as a tool in revealing the graph's structure.

The purpose of this Report is to describe the algorithm introduced by Szirmay-Kalos and discusses the utilization of this algorithm in three dimensions.

Acknowledgements

When I started to do this report, I had no idea that research in graph drawing has already grown so advanced. I would like to take this opportunity to express my sincere gratitude to my supervisor, professor Peter Grogono, for giving me the chance to dive into this exciting field, for all his ideas, guidance and support throughout the course of this report. Most importantly, professor Peter Grogono provided me with the useful resources for the report. Best wishes goes to professor Hovhannes Harutyunyan for reviewing my report.

I would also like to thank the anonymous students for their encouragement and thoughtful comments through the period of my studying. My special thanks are also due to all the professors and staff in the computer science department for their endless support during my studies.

I gratefully thank my dear parents and sisters for their love, their help and their support. Finally, but most significantly, I dedicate this to my beloved husband Julian Cheng and my two wonderful daughters Sophie Cheng and Juliana Cheng for being the greatest family one could have, without their extraordinary support, this work would not have been done.

Table of Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

1.1 Introduction

Many programs designed for interactive applications, scientific visualization, multimedia etc., should draw or display graphs on the computer screen. For visual presentations layout criteria center on the topological relationships of the graph, the geometrical position of the nodes is not that important, but the meaning of the diagram should be conveyed to the viewer quickly and clearly. Usually the following criteria are expected to be the features of an easily understood graph [24]:

- Be in the center of the display area and expand loosely over the available area.
- Minimize the number of edge crossing and have enough space around nodes to provide some readable information.
- Place linked node groups in a compact manner and put unrelated nodes as far away as possible.

- Have a "natural" arrangement (similar to those human observers are used to).

Unfortunately, these subjective criteria are almost impossible to be formulated by mathematics. However, they can be formulated as optimization goals for the graph drawing algorithms [27]. Sometimes, these aesthetics are such that optimality of one may prevent optimality in others. Additionally, graph layout algorithms in general can be viewed as optimization problems and are typically NP-complete or NP-hard. These observations suggest a heuristic approach to general graph drawing for many applications [26].

The dynamic layout algorithm [24] is an analogous approach, which applies mechanical system theory [25] to provide the solution for the "natural" arrangement problem, combined with heuristic methods to add those features that are not reflected by the behavior of mechanical systems and also to reduce the response time of the algorithm.

This algorithm is one of the force-directed methods. It simulates a mechanical system in which edges in the graph are modeled as springs and vertices are modeled as rings connecting edges (springs) incident on a vertex. Then it aims to have the system reach a minimum-energy layout of the vertices. Allowing the system to reach its equilibrium in three dimensions would enable the use of interactive computer visualization as a tool in revealing the graph's structure [26]. This paper reports the extension of the dynamic layout algorithm from two to three dimensions. The implementation and paper are closely following the approach and explication of Szirmay-Kalos [24].

1.2 Basic Concepts for Graphs and Graph drawing

To improve the understanding of this general graph drawing algorithm, I introduce some basic concepts related to the graph drawing. These concepts can be found in many textbooks and here are from [1].

A *graph* $G = (V, E)$ consists of a finite set V of *vertices* and a finite multi-set E of *edges*, that is, unordered pairs (u, v) of vertices. The vertices of a graph are sometimes called *nodes*; edges are sometimes called *links*, *arcs*, or *connections*.

An edge (u, v) with $u = v$ is a *self-loop*. An edge that occurs more than once in E is a *multiple edge*. A *simple graph* has no self-loops and no multiple edges. The algorithm described in this report deals with simple graphs.

The *end-vertices* of an edge $e = (u, v)$ are u and v; we say that u and v are *adjacent* to each other and e is *incident* to u and v. The *neighbors* of v are its adjacent vertices. The *degree* of v is the number of its neighbors.

A *directed graph* (or *digraph*) is defined similarly to a graph, except that the elements of E, called *directed edges*, are ordered pairs of vertices. The directed edge (u, v) is an *outgoing edge* of u and an *incoming edge* of v. Vertices without outgoing (resp. incoming) edges are called *sinks* (resp. *sources*). The *indegree* (resp. *outdegree*) of a vertex is the number of its incoming (resp. outgoing) edges.

A graph $G = (V, E)$ with n vertices may be described by a $n \times n$ *adjacency matrix* A whose rows and columns correspond to vertices, with $A_{uv} = 1$ if $(u, v) \in E$ and $A_{uv} = 0$ otherwise as shown in Table 1, or it can be described by giving a list L_u of edges incident to vertex u for each $u \in V$ as in Table 2.

	1	2	3	4	5
1	0	1	1	1	1
2	1	0	1	1	0
3	1	1	0	1	0
4	1	1	1	0	1
5	1	0	0	1	0

Table 1: An adjacency matrix for graph G_1 shown in Figure 1 (a)

L_1	(1,2)	(1,5)
L_2	(2,3)	
L_3	(3,4)	
L_4	(4,5)	(4,6)
L_5	(5,2)	
L_6	(6,3)	

Table 2: Adjacency list for digraph G_2 shown in Figure 2 (b)

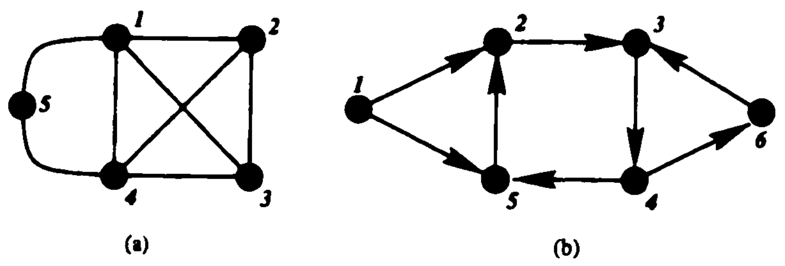

Figure 1: (a) A drawing of graph G_1 and (b) A drawing of directed graph G_2

Figure 1 above shows the drawing G_1 for graph in Table 1 and the drawing G_2 for graph in Table 1. Here we need to mention that a graph and its drawing are different concepts. A graph can have many different drawings according to the definition of a drawing. A definition of a drawing of a graph in the simplest form is as following [1]: a *drawing* Γ of a graph G is a function which maps each vertex v to a distinct point $\Gamma(v)$ and each edge (u, v) to a simple open Jordon curve $\Gamma(u, v)$, with endpoints $\Gamma(u)$ and $\Gamma(v)$.

A drawing Γ is *planar* if no two distinct edges intersect. A graph is *planar* if it admits a planar drawing [1]. From the experience in our daily life, we know that edge crossing reduce readability [13] of the graph. Therefore in graph drawing, we need to apply the theory of planar graphs as much as possible, since this well-developed theory [28] can be used to greatly simplify topological concepts. Also we need to know that planar graphs are "sparse": Euler's formula [29] implies that a simple planar graph with n vertices has at most $3n - 6$ edges. For large n, this is much less than the maximum number of edges, which is $n(n-1)/2$. Thus most graphs with many vertices are either non-planar or have few edges.

Finally, when drawing a directed graph, the underlying undirected graph can be constructed by forgetting the directions of the edges and then the directed edges can be drawn as arrows. Thus the terminology of graphs can be applied to digraphs too.

1.3 Problem Statement

The dynamic layout algorithm takes a graph $G = (V, E)$, V is a set of vertices and E is a set of pairs of elements of V and the graph is undirected. But the graphic program should be able to show the directions of edges if the graph is directed. The task of the algorithm is to determine the positions of all the vertices of graph G in the display area and the resulted layout of the algorithm should be the subject to the primary aesthetic criteria stated in Section 1.1.

The graphic program should have two ways to display the graph. First, it displays the graph randomly by assigning coordinates of vertices randomly. Second, it should display the graph according to the subjective criteria of graph drawing after applying the algorithm. It displays each vertex as a spheres (or cubes, ovals) and each edge as a thin directed cylinder joining two vertices. The user should be able to move and rotate the graph to see it better.

1.4 Report Organization

This report is organized as following: the first section talks about basic concepts for graphs and graph drawing and problem statement of the algorithm. Section 2 discusses fundamental issues in graph drawing and introduces the most general and easy techniques of the best-known force-directed model for graph drawing. Section 3 describes the dynamic layout algorithm used in this system. The design of the mechanical system to simulate the algorithm is presented in Section 4. The purpose of Section 5 is to explain the detailed implementation of the algorithm and to discuss some problems met during implementation. Section 6 presents the results attained in several aspects. Finally, Section 7 gives concluding remarks and future enhancements.

Chapter 2

BACKGROUND AND RELATED WORK

This section introduces the background of force-directed graph drawing algorithms and overviews some more popular algorithms in comparison to the dynamic layout algorithm.

Researchers have done a lot of hard work to define the aesthetic criteria of a graph drawing as much as possible in the proposed algorithms. One of the most important challenges in the graph drawing algorithms is to efficiently give the topology of the graph without higher computation cost. In general, graph drawing algorithms can be grouped according to several approaches. The algorithms in each approach can work only or better on graphs belonging to specific classes [1] and also are confined to specific *drawing convention*, *aesthetic*, *constrain*. Since the dynamic layout algorithm described in this paper is grouped in force-directed approach, we focus here on the main idea of force-directed approach.

2.1 Background of the Force-Directed Approach

Force-directed algorithms view the graph as a virtual physical system, where the nodes of the graph are bodies of the system. These bodies have forces acting on or between them [30]. The algorithm seeks a position for each body such that the sum of the forces on each body is zero, which means the system is in a state with locally minimal energy.

Force directed algorithms are suitable methods for creating straight-line drawings of undirected graphs [1]. Roughly speaking, a forced directed algorithm works like this: it takes an arbitrary input graph, computes the forces acted on each node according to the methods used to defined forces in this system, and then the system oscillates until it stabilizes at a locally minimum-energy configuration. There are two components in this approach: a *force model* and an *algorithm*.

- *The force model*: A model that contains the vertices and edges, which is defined by some physical forces acted on the graph.
- *The algorithm*: A technique for finding a locally minimum force state of the model, that is, a position for each vertex, such that the total force on every vertex is zero. This state defines a drawing of the graph [1].

Thus the force directed algorithms differ in two ways: one is the different force model used and another one is the different algorithm applied to find an equilibrium or minimal energy configuration.

Usually, the model contains the information for aesthetic criteria. The forces are defined so that an equilibrium configuration sashes these criteria.

Force-directed algorithms are widely used since they have two advantages. First, the physical phenomena exist in real would, so they can be understood easily and be relatively simple to implement. Second, in general, they can produce a nice-looking graph since a variety of constraints can be defined for aesthetic purposes. They often give highly symmetric drawings, and tend to distribute vertices evenly.

The simulation of a virtual physical system for object placement pre-dates the development of force directed algorithms for graph drawing [31]. Eades [9] first proposed a heuristic algorithm for drawing undirected graphs in two dimensions, based on a virtual physical model. This model is now referred to as the famous "spring model", since in this model each node is replaced with a steel ring and each edge with a spring. More recently, a lot of force-directed algorithms have been proposed and tested ([9], [10], [30], etc.). Many of them are based on the spring model.

Below are some well-defined force directed models.

2.2 Spring and Electrical Forces Model

The most popular and simple force model uses a combination of *spring* and *electrical* forces, which usually is the underlying model for other models. In this model edges are modeled as springs, and vertices are equally modeled as charged particles which repel each other. Precisely speaking [1], the forces acted on vertex v is

$$F(v) = \sum_{(u,v)\in E} f_{uv} + \sum_{(u,v)\in V\times V} g_{uv} \tag{2.1}$$

where f_{uv} is the spring force exerted on v by the spring between u and v, and g_{uv} is the electrical repulsion exerted on v by the vertex u.

Battista *et al.* have described the basic idea behind this algorithm in [1]. They follow Hooke's law to define the force f_{uv}, which is f_{uv} is proportional to the difference between the distance between u and v and the zero-energy length of the spring. The electrical force g_{uv} follows an inverse square law.

It uses three parameters l_{uv} (the natural length of the spring), $k_{uv}^{(1)}$ (the stiffness of the spring), and $k_{uv}^{(2)}$ (the strength of the electrical repulsion) to control the appearance of the drawing. It satisfies two important aesthetics: the spring force is used to guarantee the desirable distance between u and v, and the electrical force ensures that the two related vertices are not too close.

According to this model, there are several algorithms proposed to find its minimum force state. For example: Eades' [9] heuristic algorithm works with this model; Eades also notes that using the spring model can obtain an indirect benefit, which is that all edges typically have relatively uniform length and the drawing tends to be symmetric.

Quigley [30] proposed an algorithm that is based on a spring and electrical forces model, but he has a more efficient way to find the locally minimum energy layout of the system. The algorithm is named as FADE2D (Force Directed Algorithm by Decomposed Estimation 2-Dimension) that can also extend to FADE3D. It has complexity O($n \log n$), which means large graphs can be laid out using this algorithm. The main idea of the FADE algorithm is to take the initial graph layout and perform a geometric clustering of the location of the vertices, which is conducted by recursively decomposing the space. Thus it allows us to approximate the non-edge forces in a force directed graph drawing algorithm. FADE improves the performance of force directed algorithms by computing forces using a recursive decomposition of the location of the nodes rather than all the nodes directly.

2.3 The Tutte Model

The Tutte model [20, 21] differs from the model described in the previous section in three ways. First, it defines that the spring has $l_{uv} = 0$. Second, the stiffness parameter of the spring is equal to one for every edge in the graph. And third, there are no electrical forces acting on each vertex by other vertices. Thus force $F(v)$ can be simply expressed as

$$F(v) = \sum_{(u,v)\in E} (p_u - p_v) \tag{2.2}$$

Through the comparison with the previous model, we can see that the drawing with this model is not a good one, since two important aesthetics mentioned above cannot be satisfied. In order to obtain the desirable drawing, the vertex set V is divided into two sets: one contains fixed vertices and another one contains movable vertices. Therefore, to have a drawing of the graph, we need only to resolve the assignments of positions of the movable vertices. Tutte proposed a technique to solve this problem by placing each free vertex at the barycenter of its neighbors, which is the position where its coordinates are the average of the coordinates of its neighbors. Thus this technique is also named the *barycenter method*, which is one of the earliest graph drawing methods.

2.4 Graph Theoretic Distances Model

This Graph Theoretic Distance Model was pioneered in [12] and developed independently in [10]. It uses Euclidean distance to model graph theoretic distance and works only on a connected graph that is there exists a path between u and v for each pair (u, v) of vertices [1].

The main idea behind this model is: the graph theoretic distance $\delta(u,v)$ is defined to be the number of edges on a shortest path between u and v. Then the algorithm seeks a drawing of the graph such that for each pair (u, v) of vertices, the Euclidean distance $d(p_u, p_v)$ between u and v is approximately proportional to $\delta(u,v)$ between all pairs u and v of a connected graph G. Therefore the force of system is proportional to $d(p_u, p_v) - \delta(u,v)$ between vertices u and v.

Kumar and Fowler [26] extend the algorithm proposed by Kamada and Kawai [10] from positioning nodes in a plane to positioning nodes in three dimensions. The algorithm considers the energy of the system rather than the forces. It obtains a balanced layout by decreasing the energy of the system to a minimum. In order to get the minimum energy η, the following condition must hold:

$$\frac{\partial \eta}{\partial x_v} = 0, \frac{\partial \eta}{\partial y_v} = 0, \frac{\partial \eta}{\partial z_v} = 0, \forall v \in V .$$

That means the partial derivatives of η, with respect to each variable x_v, y_v and z_v, should be zero. These 3n nonlinear equations must be solved, where n is the number of the vertices in the graph. Usually they are solved using an iterative approach. During each iteration, the vertex on which has the largest force acting is moved to a position with minimized energy, while all other vertices remain fixed.

There are two major steps in this algorithm. One is to find graph theoretic distance and another is vertex position algorithm. According to Kamada and Kawai [10], the running time for the first one is O(n^3) and for the second step is O(n) as mentioned by Kumar and Fowler [26].

2.5 Magnetic Fields Model

This Magnetic Fields Model proposed by Sugiyama and Misue [15, 16] has a global magnetic field that acts on the system and also some or all of the springs are magnetized. Using the magnetic field we can control the orientation of the edges and thus the model can satisfy a greater number of aesthetic criteria than the models of previous sections.

Three basic types of magnetic fields are shown in Figure 2 [1]:

- *Parallel*: All magnetic forces operate in the same direction.
- *Radial*: The forces operate radically outward from a point.
- *Concentric*: The forces operate in concentric circles.

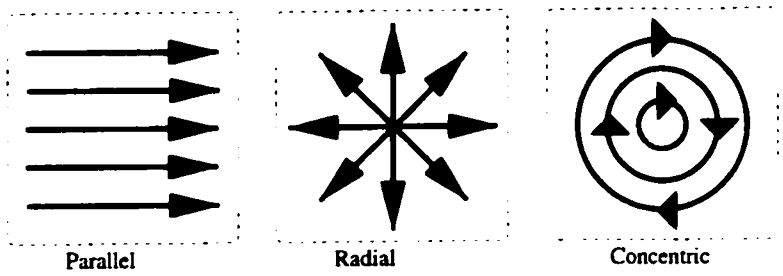

Figure 2: Types of magnetic field

These three basic fields can be used together on a force system. For example, orthogonal edges can use a parallel magnetic field in the horizontal and vertical directions.

A magnetic spring is shown in Figure 3. There are two types of magnetic springs. One is an unidirectional magnetized spring that tends to align with the direction of the magnetic field. Another one is a bidirectional magnetized spring that tends to align with the magnetic field, but in either direction.

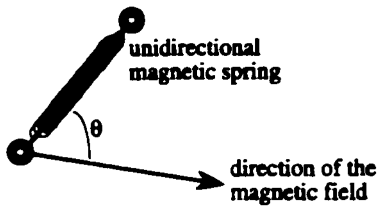
unidirectional magnetic spring
θ
direction of the magnetic field

Figure 3: Magnetic spring

Algorithms applied for this model are the same as for the spring model. That is first placing vertices initially at random location, and then at each iteration the vertices move to positions with a lower energy.

The magnetic spring model is more frequently used to handle directed graphs. We can find this model is useful in applications like tree drawing. Figure 4 shows a parallel vertical magnetic field combined with unidirectional magnetic springs. The result is that the edges tend to point downward.

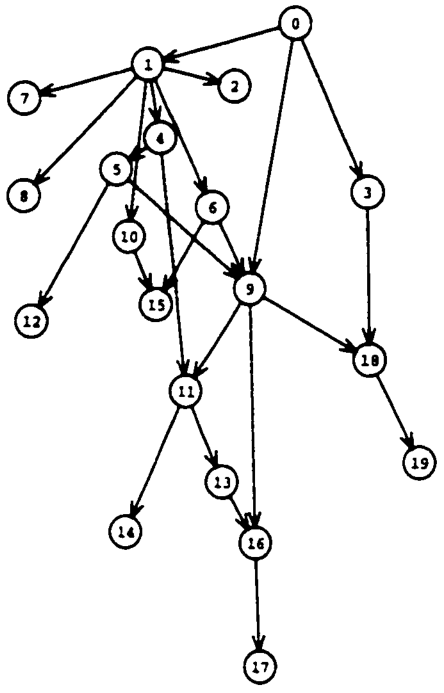

Figure 4: Magnetic spring drawing using a vertical magnetic field and unidirectional magnetic springs

2.6 General Energy Functions

Until now we have only discussed about a simple and continuous energy function η of the locations of the vertices. In order to improve the readability of a graph, we need to consider as many aesthetic criteria as possible. Some of the important aesthetics for drawings of general undirected graphs are symmetry, minimization of edge crossing and bends in edges, uniform edge lengths, and uniform vertex distribution [27]. But these aesthetic criteria are not continuous. In order to broaden aesthetic criteria in each drawing, we need to include some discrete energy functions.

Battista *et al.* [1] proposed a model, which uses an energy function that linearly combines a number of measures

$$\eta = \lambda_1 \eta_1 + \lambda_2 \eta_2 + \ldots + \lambda_k \eta_k, \tag{3.3}$$

where, for $i = 1, 2, \ldots, k$, η_i is a measure for an aesthetic criterion and λ_i is a constant. The function η_i can also be spring energy, electrical energy, and magnetic energy, as well as discrete functions for aesthetics.

In this way, a variety of aesthetics can be included by adjusting the coefficients λ_i. A large value for λ_i indicates that the *i*th aesthetic criterion is important. In general, the user can adjust the weights $\lambda_1, \lambda_2, \ldots, \lambda_i$ to suit the aesthetics of a particular application or a particular user according to the model above. Mendonca [32] has done some work on these. In his PhD thesis he shows how these coefficients can be automatically adjusted to the user's preferences without explicit user intervention.

Chapter 3

DYNAMIC LAYOUT ALGORITHM

The description of dynamic layout algorithm below closely follows Szirmay-Kalos' [24] terminology and methodology, extending his algorithm from positioning vertices in two dimensions to positioning vertices in three dimensions.

3.1 Introduction

The dynamic layout algorithm [24] is an analogous approach, which applies mechanical system theory [25] to provide the solution for the "natural" arrangement problem, combined with heuristic methods to add those features that are not reflected by the behavior of mechanical systems and also to reduce the response time of the algorithm.

The algorithm is based on a force-directed approach, since force directed approach makes sure that the wanted layout of the graph can be achieved by defining corresponding constraints. It has the features such as flexibility, ease of implementation, and the aesthetically pleasant drawing it produces.

3.2 The Dynamically Balanced Mechanical System

The algorithm models the graph as a mechanical system, where a dynamic three-dimensional structure is proposed. Vertices, v, in the graph correspond to particles, p, while edges, or absence of edges, correspond to driving forces among these particles.

General speaking, the force should be attractive between two vertices when they are not very close to each other, and should be repulsive when they are too close. In this mechanical system, whether the force is attractive or repulsive between two vertices depends on the threshold distance of the pair of vertices. If the distance of two vertices exceeds their threshold distance, the force between them is attractive, and otherwise the force is repulsive. The threshold distance of pairs of vertices is different from linked vertices to unlinked vertices. This can be easily understood, since linked vertices are expected to be close and unlinked vertices are expected to be far away. Therefore the threshold distance for linked vertices is defined as required minimum radius of a vertex to have some space for readability of vertices, and the threshold distance of unlinked vertices is defined as a parameter related to the display size and the number of vertices in the graph. To simplify the definition for these two threshold distances, the weighted edges are used, where for unlinked vertices, the weight of edges between them is zero and maximum weight of an edge is also set to control threshold distance of unlinked vertices. So the threshold distance ("constraint" in equation (3.1)) is proportional to the value of the weight:

If A and B are arbitrary objects:

$$constraint(A, B) = constraint_{\min} + (weight_{\max} - weight(A, B)) \times constraint_{\text{scale}} \qquad (3.1)$$

here the constraint scale is used to adjust value of weight to consist with constraint.

3.3 Computing a Local Force

The force exerted on a vertex A consists of three different components.

(1) The first component is the force on A due to its linked vertices. It is defined as being proportional to the difference of the actual distance of the vertices and threshold distance of the vertices. For example, the force on A due to B is:

$$\vec{force}_B(A) = (constraint(A, B) - distance(A, B))/distance(A, B) \cdot (\vec{pos}(A) - \vec{pos}(B)) \quad (3.2)$$

where $\vec{pos}(A)$ and $\vec{pos}(B)$ are the position vertices of vertex A and vertex B respectively.

(2) The second component is the force on A due to the forces of the wall of the display area boundaries. The direction of these forces always points inward and is perpendicular to the wall. It is defined to be zero if the vertex is inside and far away from the wall. In order to keep the vertices well inside the display area, some margin around the wall is defined. The force of the wall and the force of margin are different. If the vertex is inside but is in margin, the force of the wall is defined to be proportional to the distance from the wall with the force of margin, otherwise it is defined to be the combination of the force of the wall and the force of the margin. The following formulas are used to calculate the force according to three situations.

If A is inside the working area, but not in the margin:

$$\vec{force}_{wall}(A) = 0 \quad (3.3)$$

If A is in the margin:

$$\vec{force}_{wall}(A) = (margin_{wall} - distance(A, wall)) \times \vec{margin\_drive}_{wall} \tag{3.4}$$

If A is outside:

$$\vec{force}_{wall}(A) = distance(A, wall)) \times out\_out\_\vec{drive}_{wall} + margin \times \vec{margin\_drive}_{wall} \tag{3.5}$$

(3) The third component is the friction on A due to its movement. Let the coefficient of the friction be μ and the current speed $\vec{v}(t)$ in time t. Then the friction on A is proportional to μ and the speed of A's movement. The direction of this force always points to the opposite of the driving forces.

$$\vec{force}_{friction} = \vec{v}(t) \cdot \mu \tag{3.6}$$

Since the driving force

$$\vec{D} = \vec{force}_B(A) + \vec{force}_{wall}(A) \tag{3.7}$$

Then the resultant force on a vertex A is

$$\vec{F} = \vec{D} - \vec{force}_{friction} \tag{3.8}$$

3.4 Positioning vertices in Three Dimensions

The position of a vertex in a three dimensional space is represented by a triple (x, y, z), where x, y and z are the values of the X, Y, and Z coordinates respectively. The type of vertices in the graph can be fixed to some location initiated or can be moveable, which means only moveable vertices are moved according to the forces exerted on them. Thus each moveable vertex will move until the system reaches equilibrium, where all forces exerted on each vertex for all vertices and the speed of all vertices are zero or under a minimum acceptable value.

Since the forces are minimized in the stable state, linked vertices are located close to each other and unlinked vertices are located far away. Since the forces of the display area boundaries are also considered, the whole system (the graph) will be placed somehow in the middle of the display area.

The discrete time simulation is used to simulate the movement of vertices until the stable state of the system is found.

Let the mass of the vertex be m and the current position be $\vec{r}(t)$ in time t

Then Newton's law can be applied to calculate the approximate speed and position in time $t + dt$ of a vertex:

$$(\vec{v}(t+dt) - \vec{v}(t))/dt = \vec{F}/m \tag{3.9}$$

$$(\vec{r}(t+dt) - \vec{r}(t))/dt = (\vec{v}(t) + \vec{v}(t+dt))/2 \tag{3.10}$$

Placing the resultant force (3.7) obtained in previous section into (3.8), $\vec{v}(t+dt)$ and $\vec{r}(t+dt)$ can be expressed:

$$\vec{v}(t+dt) = (1-f)\cdot\vec{v}(t) + I\cdot\vec{D} \tag{3.11}$$

$$\vec{r}(t+dt) = \vec{r}(t) + (\vec{v}(t) + \vec{v}(t+dt))\cdot dt/2 \tag{3.12}$$

where f is the friction parameter: $f = \mu/m \cdot dt$ and I is the inverse inertia describing how quickly a vertex reacts to forces: $I = dt/m$.

Here f and I are also time-dependent to allow quick stabilization without excessive transients and oscillation. It can be easily understood, since the system reacts to the driving force quickly when the friction parameter f is small and the inverse inertia I is large. It is a desirable phenomenon in the beginning of the simulation but not at the end, when the system reaches the stable state, since it can result in excessive transients and oscillation around the stability point. So the following formulas are used to defined f and I:

$$f(t) = friction_{\min} + (friction_{\max} - friction_{\min}) \times t \mathbin{/} time_{\max} \tag{3.13}$$

$$I(t) = iinertia_{\min} + (iinertia_{\max} - iinertia_{\min}) \times t \mathbin{/} time_{\max} \tag{3.14}$$

where $time_{\max}$ is the maximum response time allowed in an interactive environment.

Thus the position of all vertices at time $t + dt$ can be obtained with equation (3.11) by calculating the speed of vertices that are moved under the forces exerted on them at first.

The simulation has three possible states: first, it found the stable state of the system, which means $\vec{D}$ is zero or less than a minimum value defined for all vertices; second, the force on some vertices is too strong ($\vec{D} > \vec{force}_{\max}$) so that the excessive transients and oscillation around some points happened; third, maximum response time has been exceeded without finding a stable state of the system.

The placement of the position of all vertices is iterated. During each iteration the system is checked to see if it is in one of the three states described above. If this is the case, the simulation stops and the resulted position of all vertices is the final layout of the graph for this simulation.

3.5 Initial Position

The initial positions of vertices in the system is critical, since there are many possible stable states, which means the forces on all vertices are minimum, to give good or bad layout of the graph. Another reason for careful selection of the initial positions is that the acceptable simulation time also depends on the initial positions of vertices, which means a poorly selected initial layout may need a great amount of time to reach a stable state.

In order to find appropriate initial positions, Szirmay-Kalos also proposed a heuristic method [24] which guarantees that linked vertices are close together while unlinked vertices are distant from each other, and also that the initial positions are not too far from mechanical stability view. This heuristic method places the fixed vertices first, which may be chosen randomly or be given by user, then places the movable vertices one after another. That is: calculating the center of gravity of the vertices that have a link to the new vertex and the center of gravity of unlinked vertices first, then placing the new vertex at the reflection of the center of gravity of unlinked vertices about the center of gravity of linked vertices. After placing a new vertex, the dynamic layout algorithm is applied to establish a nearly stable state of subsystem (already placed vertices). Thus the whole movement of already placed vertices decreases when every time introduces a new vertex as the number of already placed vertices increases.

3.6 Dynamic Layout Algorithm

Below summarizes the algorithm for locally minimum forces in the mechanical system in which vertices move in three dimensions. The input to the algorithm is a graph.

Input: graph $G = (V, E)$; a partition $V = V_0 \cup V_1$ of V into a set of fixed vertices and a set of movable vertices

Output: a position p_v for each vertex of V, such that the energy minimum of the graph is to be found

```
1. Set the maximum response time (maxi_time) allowed in an interactive environment
2. Find the first movable vertex in the vertices list
3. If no movable vertex is found, **return** STOPPED
4. If the first movable vertex is found, Set the speed of every movable vertex to 0
5. **Repeat**
   **for** each *dt* time step **do**
       1. Initialize force in each vertex to 0
       2. Calculate friction and response (inertia) values according to *dt*
       3. Calculate drive force between each pair of related vertices
       4. **Repeat**  add additional forces and determine maximal forces
          **foreach** movable vertex **do**
              1. Calculate drive force of boundaries and add to relation forces
              2. Move vertex by force
              3. Calculate maximum force
       5. If maximum force is less than minimum force which is considered as 0,
          **return** STOPPED
       6. If maximum force is greater than maximum force which shows instability,
          **return** UNSTABLE
   **until** maxi_time is reached, **return** TOO_LONG
```

Chapter 4

SYSTEM DESIGN

The main feature of this mechanical system is that it can apply the dynamic layout algorithm quickly and efficiently. It is designed by using object-oriented method.

4.1 Vector Data Structure Design

The *vector* data structure is designed to provide the vertex operations such as addition, subtraction and multiplication, etc. It also provides the functions as obtained the current x, y and z coordinates of a vertex in the world coordinate system. It is the basic element used in the system, since the calculation of the forces and placement of vertices need to use its basic operations.

4.2 NodeElem Data Structure Design

NodeElem data structure is designed as a vertex item in the dynamic graph data structure. The main role of the *NodeElem* is to get the next vertex item in the graph, get the relation list that contains all linked vertices to the vertex and also get the unique identity number for the vertex. It is designed in a way such that it connects the *vertex* with the *graph*.

4.3 RelationElem Data structure Design

The *RelationElem* data structure is designed to present an edge in the dynamic graph data structure. The goal of *RelationElem* is to get the next edge of the vertex, get the weight of the edge and return a pointer which points to the linked vertex with the edge.

4.4 Graph Data Structure Design

In order to apply the algorithm efficiently on a graph, a *graph* data structure plays a very important role in the system. Actually it is a dynamic graph data structure.

The *graph* is designed to represent a mechanical analogy system, which is an orthogonal list of vertices and edges. In this graph data structure, the vertices are placed on a single linked list, where fixed vertices are at the beginning with negative serial number, and movable vertices are at the tail with positive serial number. The edges of a given vertex are also stored on a linked list that connects to the given vertex. The edge object contains its name, type, intensity parameters and a pointer to the related vertex. The relation of two vertices is stored on the edge list of the vertex that has smaller serial number, which can be seen in the Figure 5 clearly.

Figure 5 is a simple graph structure diagram. In this diagram we can see that vertex node contains at least two pointers, one points to the next vertex node in the vertices list and another one points to its relation list. The node on the relation list also contains at least two pointers, one points to the next relation on the relation list and another one points to related vertex node. We can see clearly that the related vertex pointers always point to the vertex nodes that stored after the given vertex in the vertices list. The detailed structure for vertex node, relation node and graph structure can be found in Section 5.2, 5.3 and 5.4.

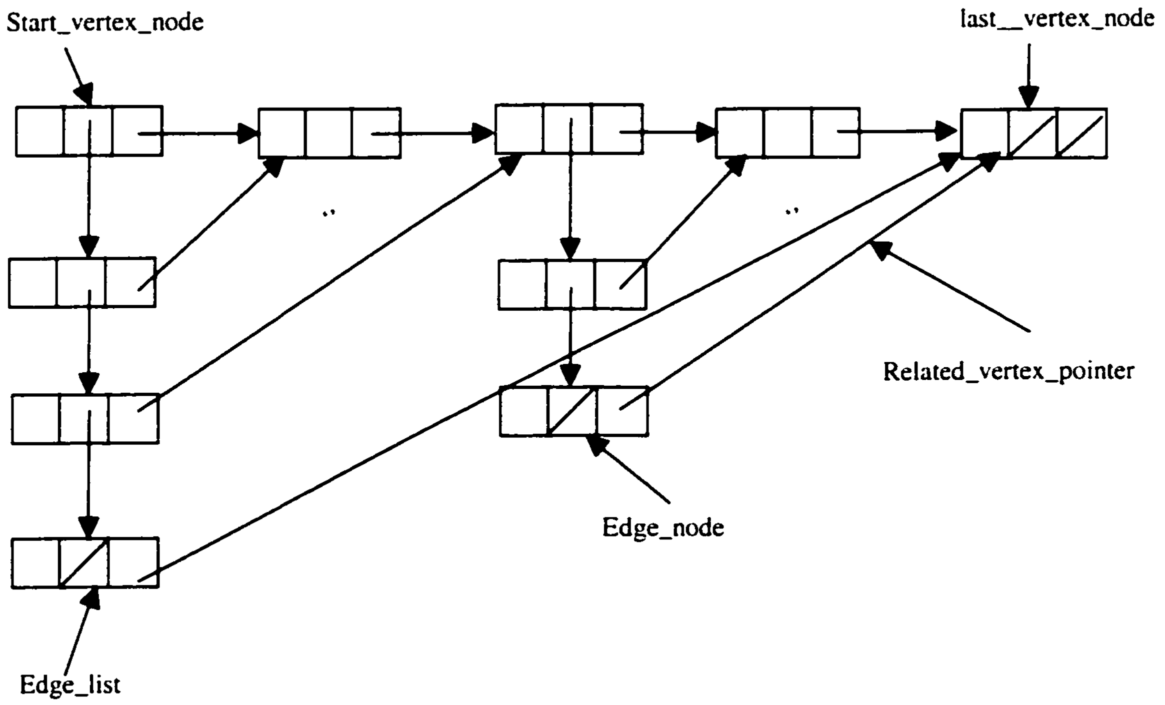

Figure 5: Graph structure for implementing the algorithm

The graph structure has all the functionality to apply the mechanical approach. The most important one is the function to apply the dynamic layout algorithm. Other functions are designed to help the implementation of the algorithm.

4.5 The Class Diagram of the System

The following class diagram (Figure 6) shows the relationship between data structures of the system.

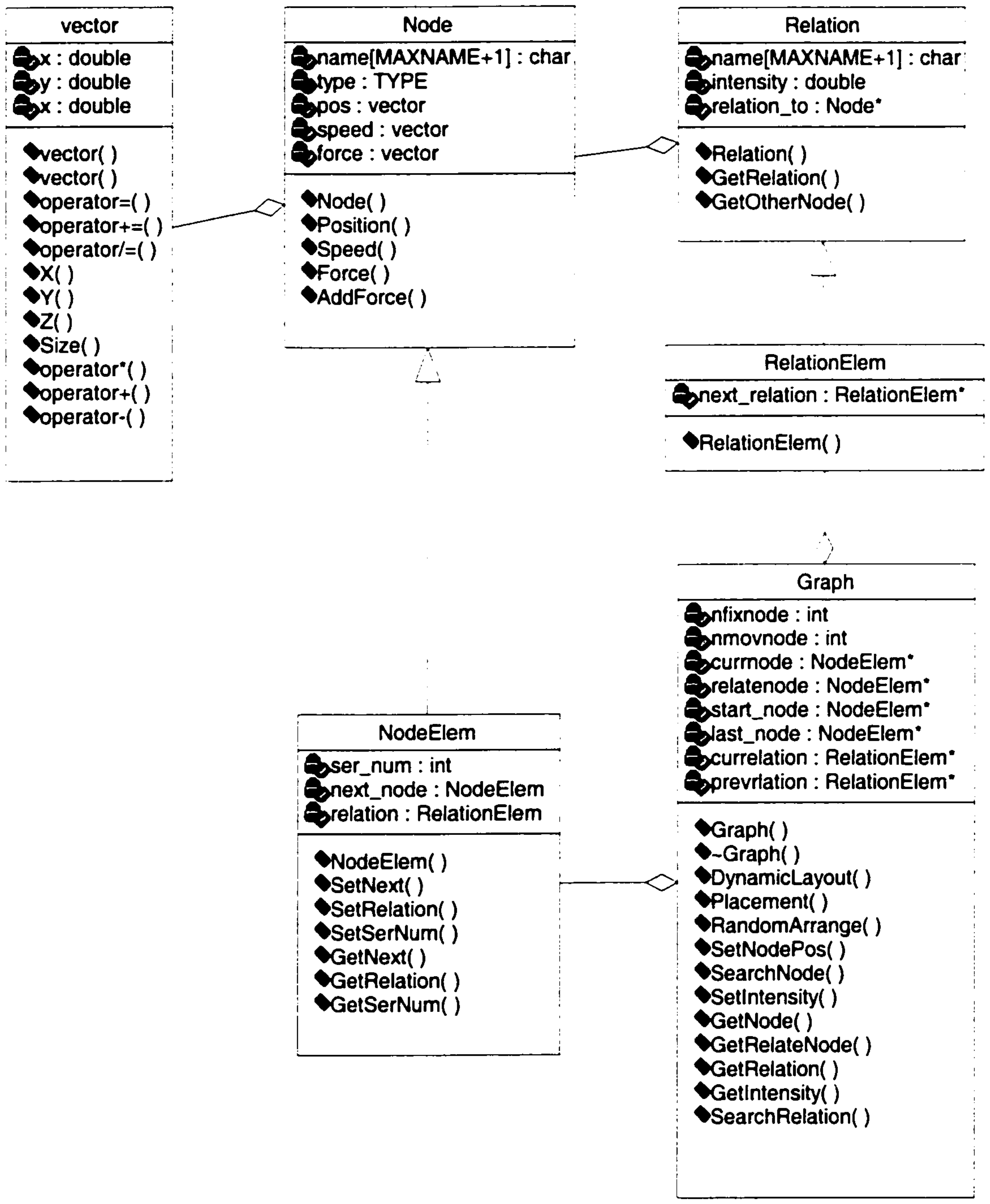

Figure 6: Class Diagram of the System

4.6 The Graphic Program Design

The graphic program used here is to display the layouts of the graph in three dimensions before and after applying the dynamic algorithm so that we can compare and observer the difference between them more clearly.

The following diagrams show the interfaces for graphic program. It contains a display window and a main window. In the main window (Figure 7) it contains all the guidance for manipulating the display of the graph. It also shows the number of iterations and the state of the system after applying the algorithm. In the display window (Figure 8) the user can click the right button on the mouse to get the menu, which gives the user the options to manipulate the perspective viewing, lighting and the shape of the vertices etc. in the graph.

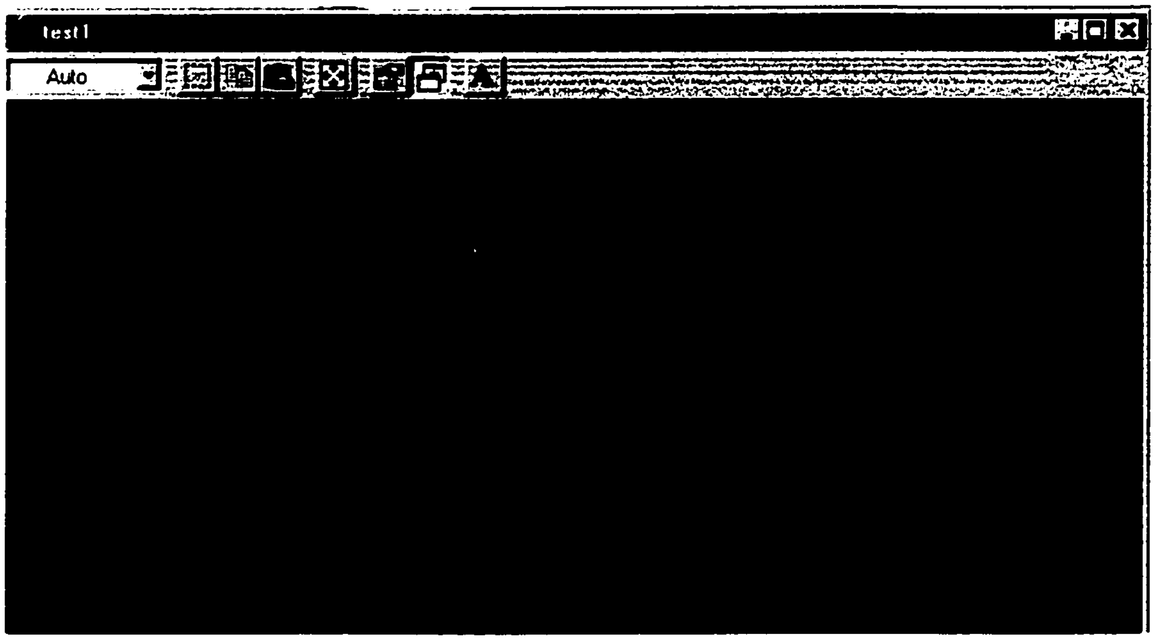

Figure 7: The main window of the graphic program

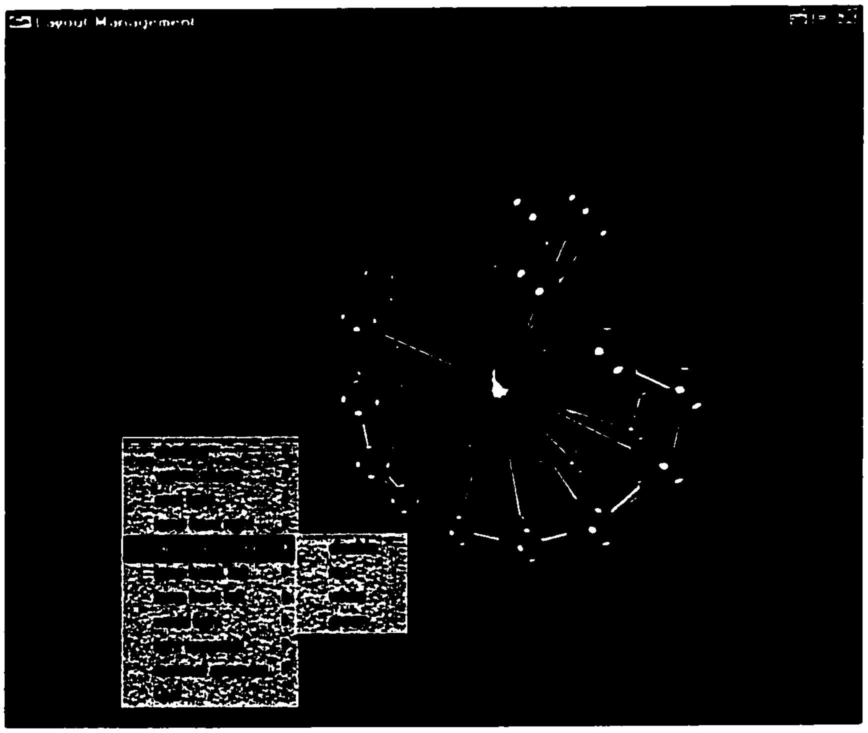

Figure 8: The display window of the graphic program

Chapter 5

THE IMPLEMENTATION

The simulation of the algorithm was written in C++ and the resulted layout of given vertices was showed by using OpenGL. Microsoft Visual C++ is the tool to implement the system and show the resulted layout. Development of this algorithm in three dimensions was based on a program written by Szirmay-Kalos for two dimensions.

5.1 Vector Class

The class *vector* has three attributes. They are x, y and z coordinates.

The main methods of class *vector* are:

- operator=(vector& a): the assignment operator
- operator+(vector& a): the addition of two vertices
- X(): return the x coordinate of the vertex
- Y(): return the y coordinate of the vertex
- Z():return the z coordinate of the vertex

- Size(): compute the actual distance of vertex to original point of world coordinate system

5.2 NodeElem Class

The class *NodeElem* inherits class Node. It has five inherited attributes to describe the status of a vertex in the graph such as the vertex type, the position, the speed and the force to this vertex. It also has two control pointers:

- next_node: pointer to next vertex in the graph
- relation: pointer to the first relation of this vertex

The main methods of class *NodeElem* are:

- Position(): return the position of the vertex
- Speed(): return the speed of the vertex
- AddForce(vector& f): add the given force to the vertex
- GetNext(): return the next vertex in the graph
- GetRelation(): return the first relation of the vertex

5.3 RelationElem Class

The class *RelationElem* inherits class Relation. It has three inherited attributes to describe the edge of the graph such as the weight of the edge and the linked vertex. It also has a pointer that points to the next relation of the vertex.

The main methods of class *RelationElem* are:

- Get Intensity(): return the weight of the edge

- GetOtherNode(): return the linked vertex

5.4 Graph Class

The *Graph* class uses the *NodeElem* and *RelationElem* classes to represent the vertices and edges of the graph. It has two attributes to represent the number of fixed vertices and the number of moveable vertices. It contains four vertex control pointers in vertex list:

- curmode – points to the actual vertex
- relatenode – another node which forms a pair with curmode for relation operations
- start_node – the beginning of the list
- last_node – the end of the list

It has two edge control pointers in relation list too:

- currelation – points to the actual relation
- prevrelation – points to the relation just before currelation on the actualrelation list

The main methods of class *Graph* are:

- SetNodePos(vertor a): sets the position of current vertex in the vertices list
- AddNode(pchar, TYPE): add the new vertex to the vertices list
- AddRelation(pchar, double): add the new relation to the relation list of the vertex with lower serial number
- SearchNode(pchar): search the vertex by its name

- RelSearchNode(pchar): search related vertex of current vertex by name. If the related node has smaller serial number than the current vertex, related vertex is set to be current vertex and current vertex is set to be the related vertex
- SearchRelation(): search for a relation between current vertex and related vertex
- RestoreNodes(pchar, pchar): Put the vertices and their relations read from vertices file and relations file into the graph data structure
- FirstNode(): select first vertex of the vertices list
- FirstMoveNode(): select first vertex whose TYPE is moveable in the vertices list
- NextNode(): select the next vertex of current vertex in the vertices list
- FirstRelation(): select the first relation in the relation list of current vertex
- NextRelation)(): select the next relation of current relation in the relation list of current vertex
- Placement(): place the vertices one by one according to initial positions' heuristic method
- RandomArrange(): place the vertices randomly
- DynamicLayout(int): place the given number of vertices at the beginning of vertices list by using dynamic layout algorithm
- Compare(): replace all the vertices in the graph from their random positions by using dynamic layout algorithm

5.5 Problem and Solutions

When I implemented the system, I encountered two problems that involved with the display of the graph, for example, keeping the track of some specific vertices while the graph is manipulated by the user, and the efficiency of the algorithm.

Problem 1: How could we keep track of a specific vertex in the graph display, since the user often lose track of the position of their current view point with respect to the global structure of the graph when he is navigating the graph?

Solution: At first I think about numbering each vertex so that when the graph is viewed after it was rotated, the relative position to other vertices of a vertex can be kept following. Since the number of vertices can be solved in this algorithm is random, the texturing of OpenGL is hard to be applied in this situation. There are several ways to partially solve the problem.

1. Adding a smaller secondary window showing a global overview with the current view location marked can provide some guidance to the user
2. Put numbers in some vertices, then it helps to follow the relative positions of vertices, especially when the number of vertices in the graph is moderate
3. Texturing the fixed vertices or moveable vertices so that they can be distinguished easily from each other. I used this solution in this Report.

Problem 2: How to know the total number of iterations after the system to reach the stable state?

Solution: Declaring a global variable to hold the number of iterations for each time the algorithm is applied and sum them, which is used with heuristic initial position method. The same variable can be used for Compare() function too, when the algorithm is applied from scratch configuration of the graph. The difference is that in Compare() function the algorithm only is applied once.

Chapter 6

RESULTS

6.1 Comparison of Three Dimensions to Two Dimensions

The dynamic layout algorithm extended to three dimensions exhibits the same ability as dynamic layout algorithm for graph drawing in two dimensions. It displays symmetries clearly in the resulted layout. In addition, a three dimensional display provides a means for overcoming some of the difficulties with two dimensions display such as false appearance of edge crossing, since it allows the graph observer to move and rotate the graph in an interactive system so that the projection of graph can be found with true edge crossing.

The main usefulness of three-dimensional graph is lying on the ability of conveying, or allowing the observer by interactive viewing to discover more and true information of the graph that may not be possible displayed in two dimensions. Below I show two graph layouts that take a same description of graph and present it both in two dimensions and three dimensions. Figure 9 is the layout of a weighted graph with 20 vertices presented in Szirmay-Kalos' paper. Using the same data the three-dimensional layout is presented from three different views with lighting and perspective. The first one is presented in Figure 10, which is the three-dimensional layout of the same weighted graph. Figure 11 is a view of three-dimensional layout from up side of the graph in Figure 10, and Figure 12 is shown the graph viewed from right side of the graph in Figure 11.

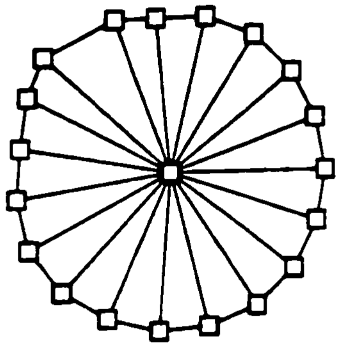

Figure 9: A weighted graph with 20 vertices in two dimensions after running the algorithm

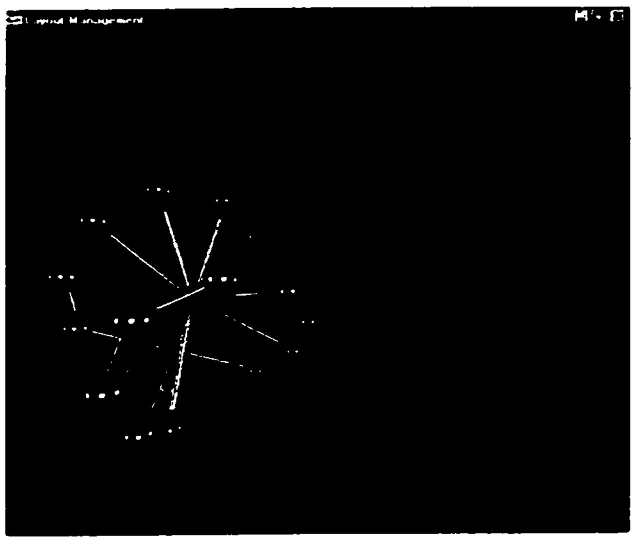

Figure 10: Three-dimensional solution of same graph in Figure 9

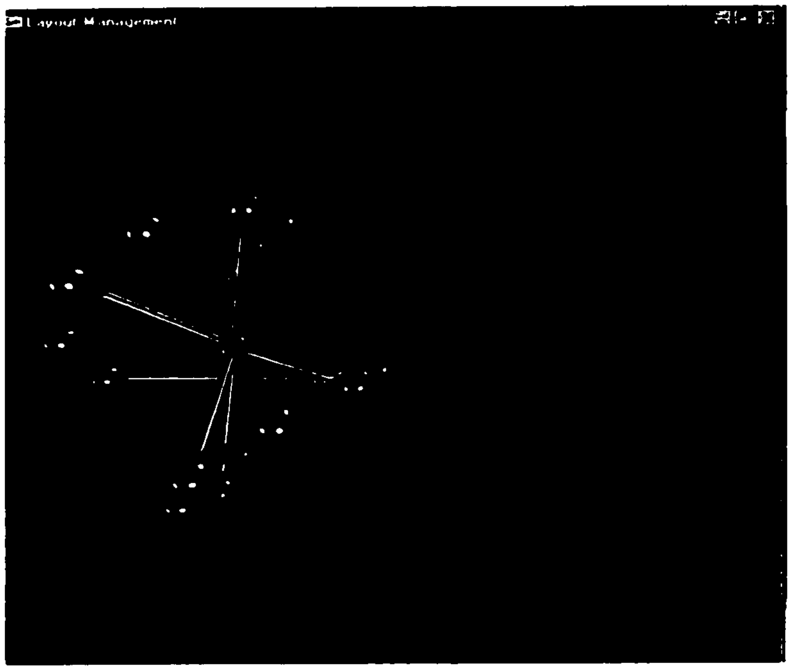

Figure 11: Three-dimensional solution of graph viewed from up side of Figure 10

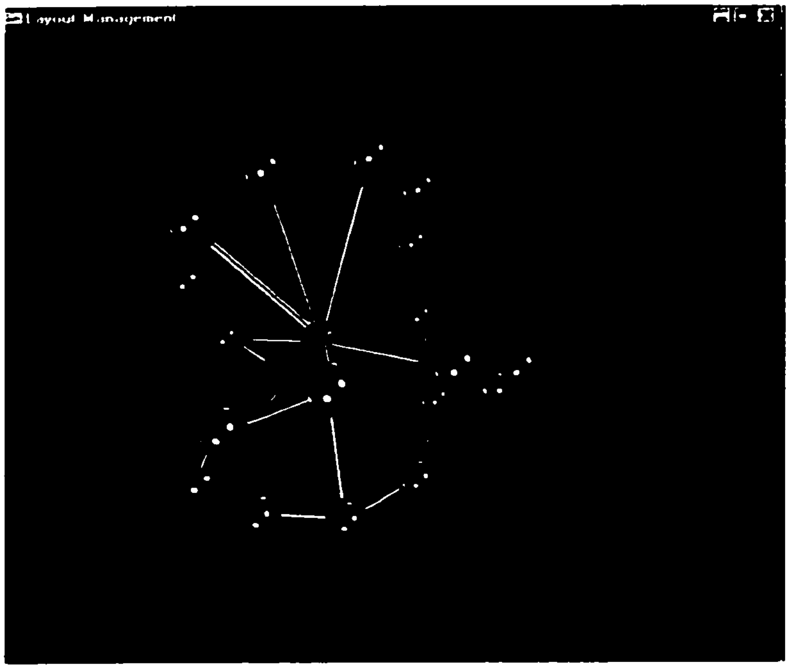

Figure 12: Three-dimensional solution of graph viewed from right side of Figure 11

Through diagrams above, we can see that the algorithm in two dimensions tries to place the vertices on circumference of a circle and the algorithm in three dimensions tries to place the vertices on the surface of a ball. This can be seen more clearly in Figure 15. It is also in accordance with local minimum energy theory that is the energy of the system is dependent on the length of elongation that the distance of two vertices underwent.

6.2 Experimental Results

I used the dynamic algorithm to assign the coordinates for two graphs. One has 30 vertices and another one has 60 vertices. It is shown in the following diagrams.

Figure 13: A sample graph with 30 vertices arranged randomly

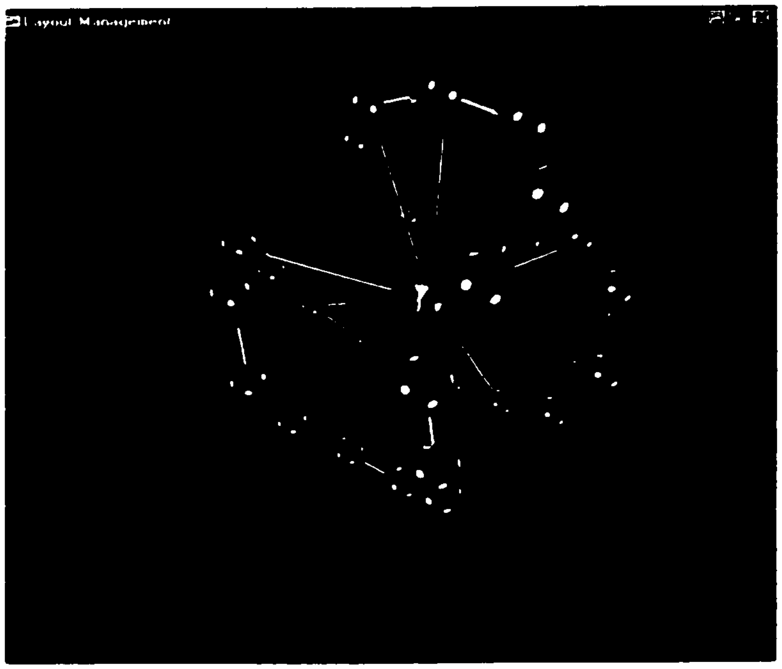

Figure 14: The same graph after applying dynamic layout algorithm

Figure 13 shows a sample graph with 30 vertices arranged randomly and Figure 14 shows the same graph after applying the dynamic algorithm with proposed heuristic initial positions method [24]. Figure 15 shows a complex graph containing 60 vertices after running the algorithm. The white vertices with red letters on them are the fixed vertices and the others are movable vertices. It is quite clear that the graph after running the algorithm is more readable and understandable than the random one, which implies that the algorithm is working.

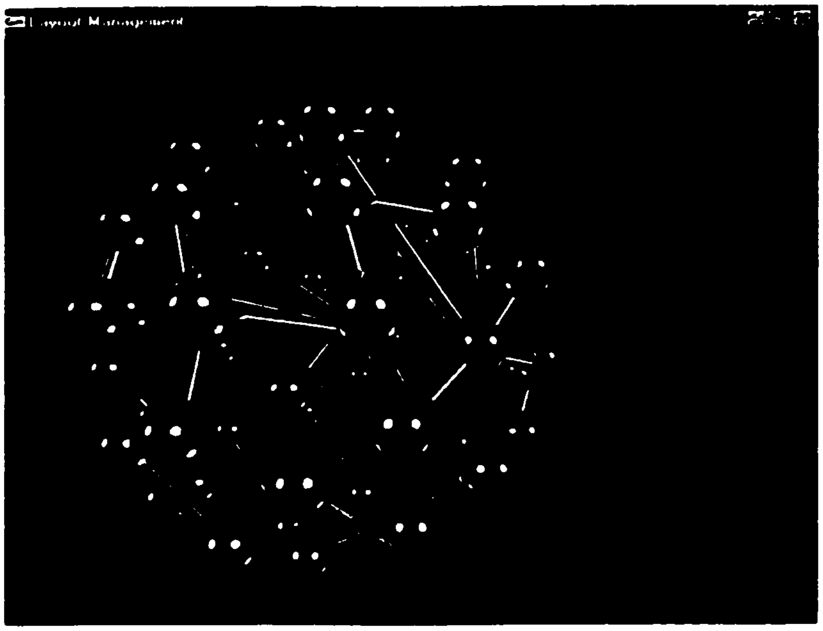

Figure 15: A complex graph with 60 vertices

There is something I also need to mention after I implemented this algorithm:

Usually when we discuss about the algorithm, we will think about its efficiency right away. The efficiency of an algorithm is measured by estimating its resource consumption. Of primary consideration when evaluating an algorithm's efficiency is the computational time required by the algorithm based on an input size. In our case here I use the number of iteration to represent the computational time. Many factors affect the computational time of the algorithm. For example, the number of vertices and edges in the graph, the initial positions of the vertices, and the used tolerance of equilibrium, which is minimum force to be considered 0 in the system. This tolerance can result slight asymmetric in the final layout of the graph (see Figure 14).

Before we apply the algorithm, Some parameters such as the value of minimum force that is considered to be 0 in this physical system, the value of maximum force that is considered to show the instability of the system, fiction boundaries and inverse inertia boundaries are needed to be set. The question here is what values we should choose for them and if they will affect readability of the graph or the computational time.

To my understanding, these parameters really influence the computational time and the final layout of the graph.

1) If we choose the value of minimum force to be bigger than the actual force exerted on any vertex, according to the algorithm, all the vertices in the system only moved once as the forces work on them, thus the computational time for the final layout of the graph is the minimum and the final layout of the graph tends to be less compact and part of it may not be as good as it should be. This can be seen in Figure 16, which is the same graph as in Figure 10 but with minimum force value set as 1, the right side of the graph is less readable.

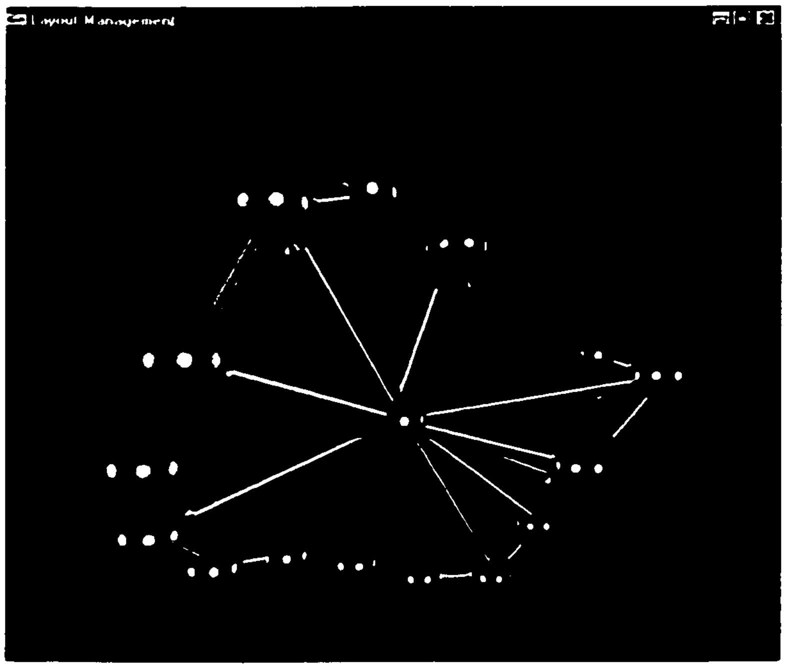

Figure 16: The same graph of Figure 10 with minimum force as 1

In Figure 10, I set the minimum force value as 0.002 and the computational time is 415 iterations. When I changed the minimum force value to 0.001, the computational time goes up to 691 iterations. Therefore a carefully chose minimum force value can result a more compact layout of the graph and have a moderate computational time.

2) The use of heuristic method of initial positions of the vertices is critical to the computational time too. I will discuss this more detailed in the next section.

To the final layout of the graph, it shows that by using heuristic method after applying the algorithm the system gives a better arrangement of the vertices than without the heuristic method. Following two diagrams show two layouts of the same graph and illustrate the some point of view. Figure 17 is the layout obtained after applying the algorithm with heuristic initial positions and Figure 18 shows the layout of the graph obtained after applying the algorithm without heuristic method. We can see clearly that the graph in Figure 18 is less readable and less beautiful.

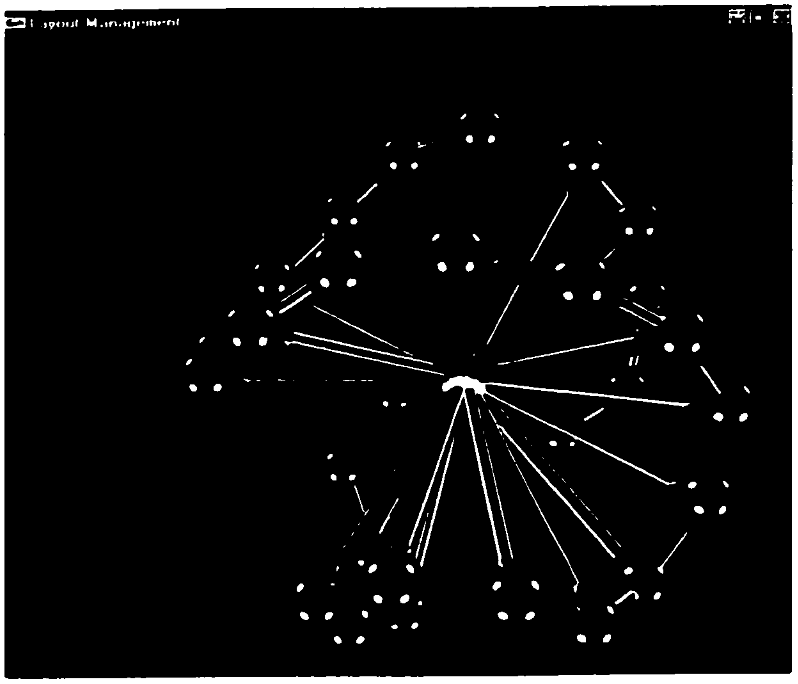

Figure 17: Graph with 30 vertices after apply the algorithm with heuristic

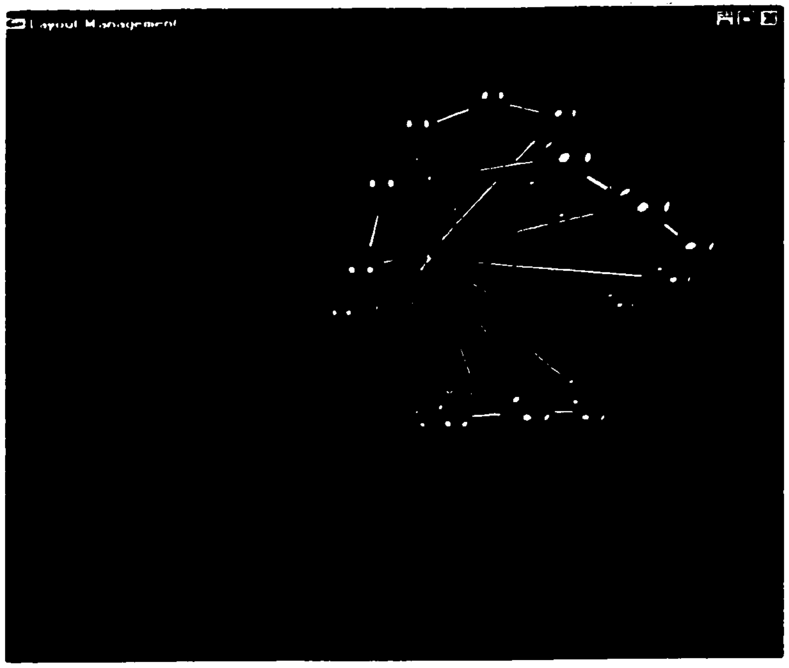

Figure 18: The same graph after applying the algorithm without heuristic

3) The number of the vertices and the number of edges in the graph is also a very important factor that affects the computational time. For example, the computational time of 20 vertices' graph with minimum force as 0.001 and heuristic method is 691 iterations, while the computational time of 30 vertices' graph with minimum force as 0.001 and heuristic method would be 1445 iterations, which is two times more than the former graph.

6.3 Related Results and Future Directions

Experiments could be conducted with more complex graph. As discussed in [24] by Szirmay-Kalos the number of total iterations, in the range of 10 to 100 vertices of the graph, is a sub-linear function of the number of vertices in the graph. The probability of finding the global minimum energy optimum decreases as the number of vertices increases. The complexity of the dynamic algorithm is $O(Tn^2)$, where T is the total number of iterations, since in a single iteration the computation required to calculate the driving force between each pair of vertices is proportional to the square of number of vertices. The algorithm is designed in such a manner that the number of iterations to be performed can be controlled effectively by augmenting the minimum force value as discussed in previous section.

If the algorithm used combining with heuristic method, the complexity can go up to $O(n^3)$, since the heuristic method places each vertex step by step with the dynamic layout algorithm. But the final layout of the graph can have partial optimal even when the system stopped before reaching the stable state because of the response time constraints.

So this allows the system to manipulate the graphs with more than 50 vertices. The example like this is shown in Figure 13.

6.4 The Knowledge I Learned From This Project

I have learned a lot from this project in various ways. When I took this graphic project as my major report, I didn't really understand the objective and meaning of it, since I did not have the enough fundamental knowledge of the graph drawing. After talking with Professor Peter Grogono and reading a lot of related materials, I knew exactly what the graph drawing is and what problems a specific algorithm is going to solve.

The force-directed approach is a popular choice for general graph drawing, since its intuitive clear analogy of a mechanical system with attracting forces along the edges and magnetic-style repelling forces for unlinked vertices. Vertices are places in initial positions and as the mechanical system works to a stable state, at which the forces exerted on a vertex for all vertices in the system are minimum, vertices are replaced to provide a nice-looking graph drawing.

From this project, I know exactly what graph drawing is, what applications it is useful for and how an algorithm worked to reach the requirement of the application. For finding an algorithm first we need to know what kind of graph we are dealing with (eg. Series-parallel digraphs, planar acyclic digraphs and orthogonal drawings, etc.); then what aesthetics the graph need to apply; and finally what techniques to use to have the graph had those aesthetics as much as possible.

Chapter 7

CONCLUSION

7.1 Conclusion

This Report introduced a dynamic layout algorithm and extended it from two dimensions to three dimensions. The algorithm takes an input of graph, uses mechanical system theory to provide a solution of positioning the vertices in three dimensions in a way such that the local forces exerted on a vertex for all vertices is minimum. Although the algorithm does not have any explicit techniques to eliminate the edge crossing, it is quite effective in doing so since it minimizes the length of edges and keeps unlinked vertices far from each other [24].

The dynamic layout algorithm extended to three dimensions exhibits the same ability to clearly display symmetries and make linked vertices in a compact way as the algorithm in a two dimensions. The extension of the algorithm to three dimensions affords one means to enhance viewing in an interactive graph visualization system. Such system would allow the user to get more information of final layout of the graph and provides an enriched display mechanism compared to the typical two-dimensional static display.

7.2 Future Work

The algorithm can be regarded as the base technique to provide a solution for the layout of a given graph. It can be combined with other techniques of graph drawing to reach the goal of the specific applications. The following are some suggestions for future work.

- Since in this algorithm, minimization of edge crossing is not considered explicitly and it is obtained by the side effect of the algorithm. So one possible enhancement to the algorithm for minimization of edge crossing can be proposed like this: the edges in the graph are also labeled, and the midpoints of the edges also have some repulsive force so that it can repel other vertices and other edges from the midpoints of the edges. Thus it would minimize the edge crossing.
- The algorithm can be used combining with magnetic fields introduced in chapter 2 to handle directed graphs. Such applications are widely used in software engineering design. One obvious example is the graph data structure in this Report. We can use two parallel magnetic fields, one horizontal and one vertical, as well as unidirectional magnetic springs. This ensures that the drawing of the graph has a tendency to be orthogonal, and the result is close to an horizontal-vertical drawing.

BIBLIOGRAPHY

[1] Giuseppe Di Battista, Peter Eades, Roberto Tamassia and Ioannis G. Tollis, *graph drawing: Algorithms for the visualization of graphs*, New Jersey: Pentice-Hall, Inc., 1999.

[2] J. Branke, F. Bucher and H. Schmeck, *Using generic algorithms for drawing unidirectional graphs,* In J. T. Alander, editor, Proceedings of the Third Nordic Workshop on Generic Algorithms and their Applications (3NWGA), pp. 193-205, 1997.

[3] C. Batini, L. Fulani and E. Nardelli, *What is a Good Diagram? A Pragmatic Approach*, In Proc. 4th Internat. Conf. On the entity Relationship Approach, 1985.

[4] C. Batini, E. Nardelli and R. Tamassia, *A layout Algorithm for Data-Flow Diagrams*, IEEE Trans. Softw. Eng., SE-12, no. 4, 538-546, 1986.

[5] M. J. Carpano, *Automatic Display of Hierarchized Graphs for Computer Aided Decision Analysis*, IEEE Trans. Syst. Man Cybern., SMC-10, no. 11, 705-715, 1980.

[6] R. Davidson and D. Harel, *Drawing Graphics Nicely Using Simulated Annealing*, ACM Trans. Graph., 15, no. 4, 301-331, 1996.

[7] G. Di Battista and R. Tamassia, *Algorithms for Plane Representations of Acyclic Digraphs*, Theoret. Comput. Sci., 61, 175-198, 1988.

[8] G. Di Battista, Roberto Tamassia and Ioannis G. Tollis, *Constrained Visibility Representations of Graph*, Inform. Process. Lett., 41, 1-7, 1992.

[9] P. Eades, *A heuristic for Graph Drawing*, Congr. Numer., 42, 149-460, 1984.

[10] T. Kamada and S. Kawai, *An Algorithm for Drawing General Undirected Graphs*, Inform. Process. Lett., 31, 7-15, 1989.

[11] C. Kosak, J. Marks and S. Shieber, *Automating the Layout of Network Diagrams with Specified Visual Organization*, IEEE Trans. Syst. Man Cybern., 24, no. 3, 440-454, 1994.

[12] J. B. Kruskal and J. B. Seery, *Designing Network Diagrams*, In Proc. First General Conference on Social Graphics, pp. 22-50. U. S. Department of the Census, 1980.

[13] H. C. Purchase, R. F. Cohen and M. James, *ValidatingGraph Drawing Aesthetics*, In F. J. Brandenburg, editor, Graph Drawing (Proc. GD '95), vol. 1027 of *Lecture Notes Comput. Sci.*, pp. 435-446. Springer-Verlag, 1996.

[14] E. Reingold and J. Tilford, *Tidier Drawing of Trees*, IEEE Trans. Softw. Eng., SE-7, no. 2, 223-228, 1981.

[15] K. Sugiyama and K. Misue, *Graph Drawing by Magnetic-Spring Model*, J. Visual Lang. Comput., 6, no. 3, 1995, (special issue on Graph Visualization, edited by I. F. Cruz and P. Eades).

[16] K. Sugiyama and K. Misue, *A Simple and Unified Method for Drawing Graphs: Magnetic-Spring Algorithm*, In R. Tamassia and I. G. Tolis, editors, Graph

Drawing (Proc. GD '94), vol. 894 of *Lecture Notes Comput. Sci.*, pp. 364-375. Springer-Verlag, 1995.

[17] K. Sugiyama, S. Tagawa and M. Toda, *Methods for Visual Understanding of Hierarchical Systems*, IEEE Trans. Syst. Man Cybern., SMC-11, no. 2, 109-125, 1981.

[18] R. Tamassia, *On Embedding a Graph in the grid with the Minimum Number of Bends*, SIAM J. Comput., 16, no. 3, 421-444, 1987.

[19] R. Tamassia, G. Di Battista and C. Batini, *Automatic Graph Drawing and Readability of diagrams*, IEEE Trans. Syst. Man Cynerb., SMC-18, no. 1, 61-79, 1988.

[20] W. T. Tutte, *Convex Representations of Graphs*, Proceedings London Mathematical society, 10, no. 3, 304-320, 1960.

[21] W. T. Tutte, *How to draw a Graph*, Proceedings London Mathematical society, 10, no. 3, 743-768, 1963. [Tut63]

[22] J. Warfield, *Crossing Theory and Hierarchy Mapping*, IEEE Trans. Syst. Man Cybern., SMC-7, no. 7, 502-523, 1977.

[23] M. Hofman, H. Langendoerfer, K. Laue and E. Luebben, *The principle of locality used for hypertext navigation*, Technical Report 1991, TU University of Brauschweig, 1991

[24] Laszlo Szirmay-Kalos, *Dynamic Layout Algorithm to Display General Graphs*, Graphics Gems IV, Academic Press, Inc., 505-517, 1994

[25] D. Tsichritzis and X. Pintado, *Fuzzy Relationships and Sffinity Links*. In Dennis Teschritzis, editor, Object Composition, pages 273-285, Universite de Geneve, Geneva, 1991

[26] Aruna Kumart and Richard H. Fowler, *A Spring Modeling Algorithm to Position Nodes of an Undirected Graph in Three Dimensions*, Technical Report 1996, university of Texas, Edinburg, Texas, 1996

[27] P. Eades and R. Tamassia, *Algorithm for Drawing Graphs: An Annotated Bibliography*, Technical Report, CS-89-09, Brown University, 1989

[28] T. Nishizeki and N. Chiba, *Planar Graphs: Theory and Algorithms*, Ann. Discrete Math., 32, 1988

[29] J. A. Bondy and U. S. R. Murty, *Graph Theory with applications*, Macmillan, London, 1976. ISBN 0-333-17791-6

[30] Aaron Quigley, *What is a Force Directed Algorithm (Spring Algorithm)?*, http://www.cs.newcastle.edu.au/~aquigley/3dfade/, 2002

[31] N. R. Quinn, Jr. and M. A. Breuer, *A forced directed component placement procedure for printed circuit boards*, IEEE Transactions on Circuit and Systems, CAS-26, no. 6, 377-388, 1979