Dynamic Parameter Spring Model for Automatic Graph Layout

Xuejun Liu

(Doctoral Program in Computer Science)

Advised by Jiro Tanaka

Submitted to the Graduate School of
Systems and Information Engineering
in Partial Fulfillment of the Requirements
for the Degree of Master of Engineering
at the
University of Tsukuba

February 2002

Abstract

A multitude of data presentation problems require the drawing or display of graphs. A graph drawing algorithm takes a graph as an input and computes a layout of the graph. The layout should be “aesthetically nice” and “easy to understand”. The Spring Model and the corresponding Spring Modeling Algorithm are well known for automatic graph layout and have been widely applied in many fields of information visualization.

In the Spring Model, the speed of layout is increased at the cost of the appearance of vibration phenomenon, which influences the stability of the algorithm and results in failure to finish the automatic graph layout. This problem of trade-off between speeding up the layout and avoiding vibration phenomenon restricts the applications of the Spring Model.

In the present work, an attempt is made to find methods to speed up the process of automatic graph layout and to avoid the existing problem. An improved spring model and its corresponding algorithm are proposed to solve the trade-off in the Spring Model. The improved model is called *Dynamic Parameter Spring Model*. In the proposed approach, *dynamic* parameters are introduced to improve the physical model. The corresponding algorithm is implemented.

We conduct performance evaluations to compare our algorithm with the Spring Modeling Algorithm. The results of evaluations show that the Dynamic Parameter Spring Model makes the process of automatic graph layout faster and more stable.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recent developments in computer science and its applications have made the visualization of complex conceptual data increasingly important. In this respect, graphs, as a simple, yet powerful and elegant data abstraction, are widely used to represent information that can be modeled as objects and connections among them. Graph drawing makes information more readable and understandable to the users.

In information visualization, graph drawing addresses the problem of visualizing structural information. Automatic graph layout raises the problem of how to get an intuitive and readable layout for a given graph. Hence, automatic graph layout algorithms are commonly used when displaying graphs because they provide a “nice” drawing of the graph without user intervention. Without automatic graph layout, the information visualization would be meaningless.

Many models and algorithms for graph layout have been proposed in recent years [1,2,3,4,5,6]. Different systems need different algorithms according to the requirements of system and application. Originally, these algorithms were designed and applied for graph drawing and automatic graph layout in the two-dimensional space. Recently, some algorithms have been designed for the three-dimensional space [7,8].

For visual presentation of undirected graphs, in these automatic layout algorithms, one of the most important issues focuses on the speed of layout. Particularly for a huge graph with a complex structure containing many nodes or vertices, the workload of computation becomes much larger and the algorithm looks heavier. As a classical and well-known Force-Directed Method, the Spring Modeling Algorithm [2] is a heuristic approach of graph drawing. It is widely applied in automatic graph layout, especially in some cases where the graph is in the status of editing. But on account of the restriction, there is a big trade-off between the speed of graph layout and the appearance of vibrations phenomenon. Based on this trade-off, in our research, we analyzed the relationship between the speed of layout and the parameters, as well as the relationship between vibrations of nodes and parameters defined in the Spring Model. The

Dynamic Parameter Spring Model and The *Dynamic Parameter Spring Modeling Algorithm* [9] are used to speed up the process of graph layout, in which the vibration phenomenon can be avoided to ensure the stability of the algorithm.

This thesis is organized as follows. Chapter 2 gives the fundamental knowledge and definitions of the concepts used in this work. Chapter 3 discusses the existing problem in Spring Model. The improved model, Dynamic Parameter Spring Model and corresponding algorithm are also described here. Chapter 4 describes the application of the Dynamic Spring Model to 3D-PP project. In Chapter 5 a discussion about the proposed approach is given. Finally, Chapter 6 presents the conclusions.

Chapter 2

Automatic Graph Layout and Algorithms

2.1 Graph Classification

As basic drawing objects, graphs can be classified as follows: **trees**, **directed graphs**, **undirected graphs**, etc. Each class of graphs can also be divided into various subclasses. There are several methods for graph classification in terms of different graphic standards. Figure 2.1 shows the graph classification by Sugiyama [10].

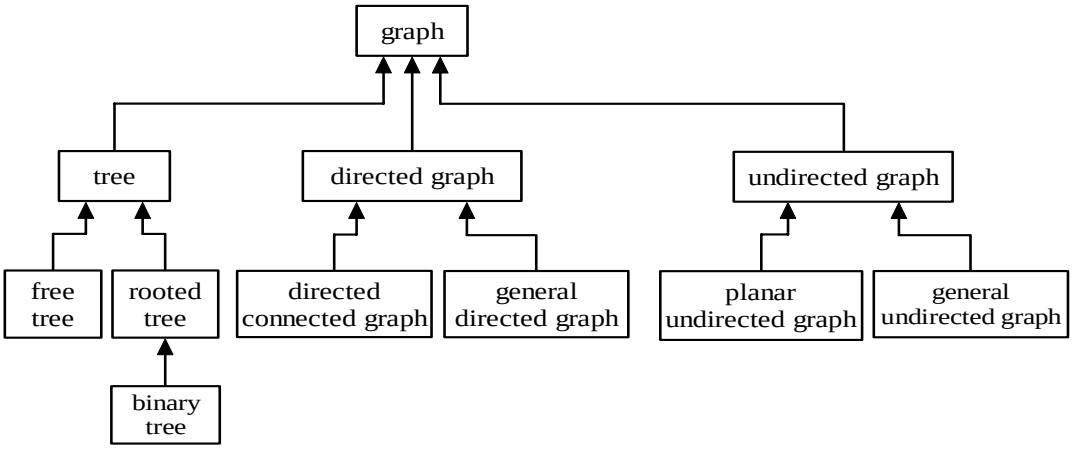


Figure 2.1 Graph classification

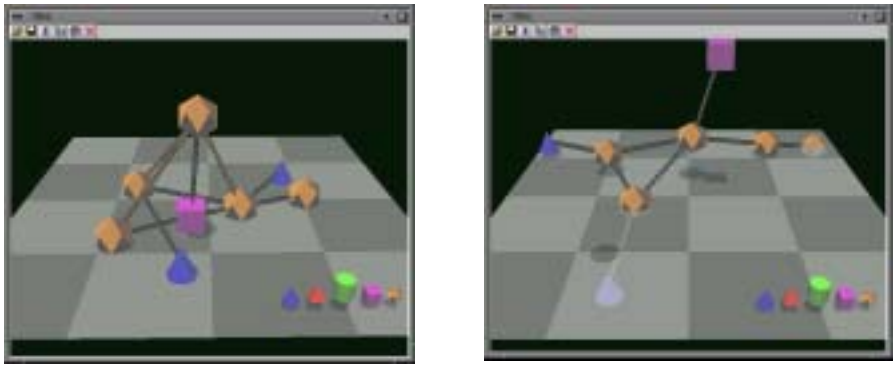
Among all the graphs, undirected graphs are the most generic graphs and are widely used in many areas of application.

2.2 Graph Drawing and Automatic Graph Layout

Graph drawing addresses the problem of visualizing structural information. More specifically, it is concerned with the construction of geometric representations of graphs and related combined structures.

Graph drawing is generally used to represent relational structures that consist of a set of entities and relationships. Such structures are modeled as graphs: the entities are vertices, and the relationships are edges. Visualization means using graphs to model the objects with the data and the dependencies between modules. A module is represented as a vertex in a graph and the dependency is represented as an edge. Visualizations of relational structures are useful to transmit the associated information to the user. Different types of graphs can be used to represent different types of structures in visualization.

Automatic graph layout (an important area of research in computer science technologies) addresses the problem of getting an intuitive and readable layout for a given graph that helps the user understand and remember the information easily. There is a wide range of applications including data structures, software engineering, database design, network design, VLSI, visual programming systems and various engineering applications.



(a) Before automatic layout **(b) After automatic layout**

Figure 2.2 Automatic graph layout

As an example, Figure 2.2 shows the difference in visualization before and after the automatic layout. After the automatic layout, the information is more intuitive and easy to understand.

Various methods and algorithms have been proposed to layout a graph automatically, such as [11,12,13] for the layout of trees, [14,15] for the layout of planar graphs, [16,17] for the layout of hierarchical directed graphs. Algorithms to draw the undirected graph automatically have also

been proposed. In this last category, Force-Directed Methods occupy an important position.

Force-Directed Method is a well-known technique for automatically laying out generic undirected graphs. Many models and algorithms have been proposed for the Force-Directed Methods, such as Spring Model by Eades [2], magnetic spring model by Sugiyama [6], and others [3,4,5,18,19].

For visual presentations of undirected graphs, one of the most important issues focuses on the speed of layout. Especially for a huge graph with a complicated structure with many nodes or vertices, the workload of computation becomes larger and the algorithm looks heavier. Therefore, the improvement of existing algorithms becomes more and more important, especially for the popular algorithm, the Spring Modeling Algorithm for automatic undirected graph layout.

2.3 Spring Model and Algorithm

2.3.1 Spring Model

The Spring Model uses a physical analogy to draw graphs. A graph is viewed as a system of bodies with forces acting between them. The algorithm for the automatic layout of graph consists of seeking a configuration of the bodies with locally minimal energy, that is a position for each body, such that the value of resultant forces on each body is zero [20].

In the Spring Model, vertices of graph are replaced with steel rings and each edge is replaced with a spring to form a mechanical system; repulsive and attractive forces are defined among rings. For example, Figure 2.3(a) shows a graph expressed by the Spring Model in its initialization configuration. After the action of repulsive and attractive forces, an equilibrium configuration is finally obtained as in Figure 2.3(b). Figures 2.3(c) and 2.3(d) show the corresponding straight-line drawing of the graph. For an undirected graph, the automatic graph layout is a process from initialization configuration to equilibrium configuration.

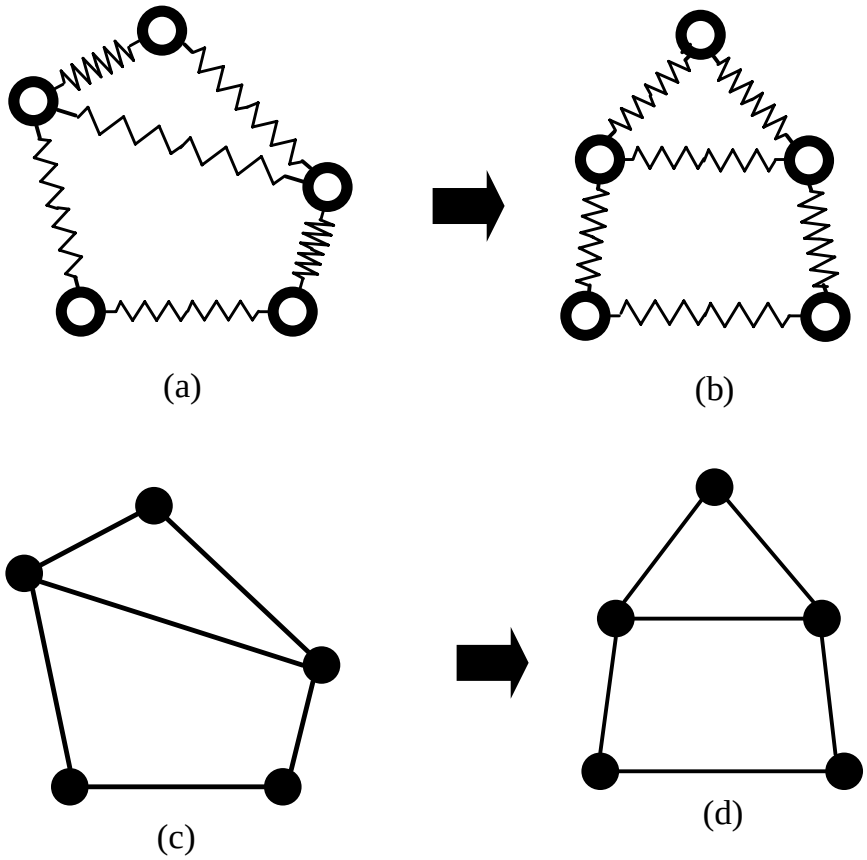


Figure 2.3 Automatic graph layout by the Spring Model

In the Spring Model, two types of forces are defined. The spring force acts on each pair of vertices (nodes) connected by spring directly. Experience shows that Hookes Law (linear) Springs are too strong when the nodes are far apart, therefore the logarithmic force are used to represents the spring force. The repulsion force is the force acting on each pair of vertices (nodes) without a direct connection. The two forces are illustrated in Figure 2.4.

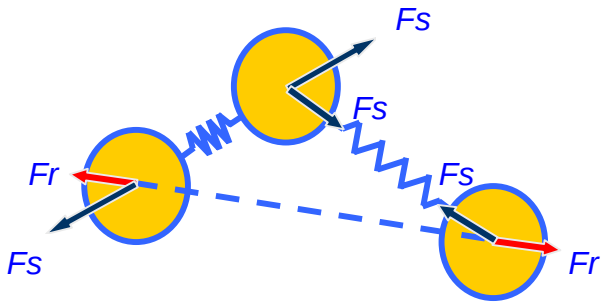


Figure 2.4 The Spring Model

In the figure above, F_s represents the spring force and F_r represents the repulsion force.

As a method of force-directed placement approach, the Spring Model has accepted four aesthetic criteria [3].

A1. Ensuring (almost) the same length for all edges.

A2. Minimizing edge crossings.

A3. Revealing symmetry.

A4. Distributing vertices evenly.

These aesthetic criteria ensure that a given generic undirected graph has a “nice” layout after the automatic layout by the Spring Model.

2.3.2 Definition of Forces in Spring Model

- Spring force

  Spring force F_s is acting on each pair of nodes connected directly in the graph. It behaves as an attractive spring force or as a repulsive spring force according to the length of spring d (the distance between two nodes).

$$F_s = C_s \log(d / C_d) \text{.....(2.1)}$$

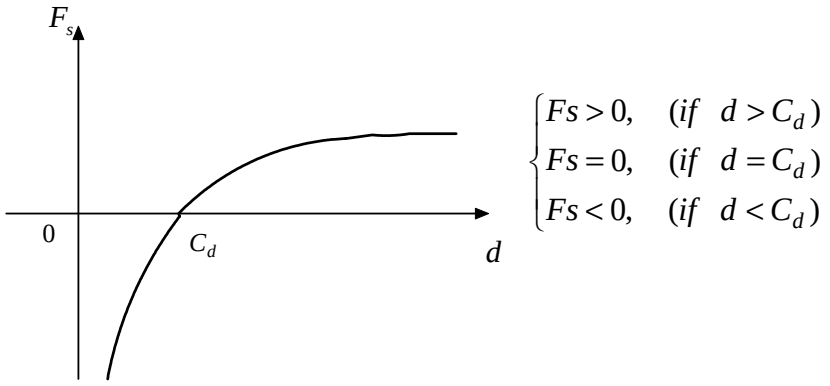


Figure 2.5 Spring force

Parameter C_d represents the natural (zero energy) length of spring. Parameter C_s and parameter C_r in the definition of repulsion force are two constant parameters in the Spring Model used to balance the influence of spring force and repulsive force. Fig. 2.5 shows the spring force.

- Repulsion force

Repulsion force F_r is acting on each pair of nodes without a direct connection by spring. It is given by

$$F_r = C_r / d^2 \quad \text{.....(2.2),}$$

where d denotes the distance between a pair of nodes. Figure 2.6 shows the repulsion force as follows.

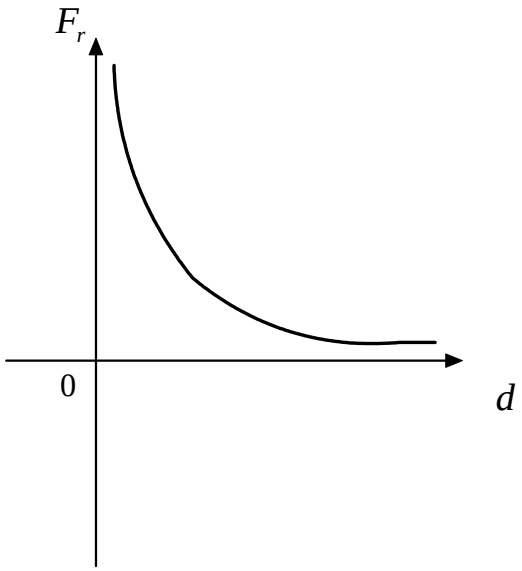


Figure 2.6 Repulsion force

For every node *i* in the graph, the acting resultant of forces is the sum of all spring forces and all repulsion forces acting on *i*.

$$F_s(i) = C_s \log\left(d(i) / C_d\right) \quad \text{.....(2.3)}$$

$$F_r(i) = C_r / d^2(i) \quad \text{.....(2.4)}$$

$$F(i) = \sum F_s(i) + \sum F_r(i) \quad \text{.....(2.5)}$$

2.3.3 Spring Modeling Algorithm

Based on the Spring Model, the Spring Modeling Algorithm for automatic graph layout is given as follows:

Algorithm:

Initialize all positions of nodes in the graph
Initialize all constant parameters
Set the stop condition.

While (stop condition not reached)
{
  for v=1 to (number of nodes)
  {
    for w=1 to (number of nodes)
    {
      if (node v and node w are a pair of nodes connected directly)
      {
        compute the spring force $F_s(v)$ acting on the node v by node w;
        $F(v) = F(v) + F_s(v)$;
      }
      else
      {
        compute the repulsion force $F_r(v)$ acting on the node v by node w
        $F(v) = F(v) + F_r(v)$;
      }
    }

    compute the movement $\delta F(v)$ of node v;
  }
  move all nodes;
}

The Spring Modeling Algorithm is a heuristic method for automatic graph layout in which positions of all nodes cannot be decided immediately. The process of layout is a simulation of force model. The simulation consists of computing the displacement of nodes by computing the resultant of force acting on nodes and moving all nodes to new positions.

Thus, in the same iteration of computing and moving, all nodes will reach their appropriate position until the graph reaches the stop condition. The stop condition needs to be set in the initialization stage according to the

requirements of the user. This process represents the process of automatic layout for a specified graph.

Chapter 3

Dynamic Parameter Spring Model and Algorithm

3.1 Problems in Spring Model

As a Force-Directed Method, the Spring Model is very suitable for automatic undirected graph layout. In real applications though, we found that there are some problems that restrict the application of algorithm.

3.1.1 User Requirements

When the Spring Modeling Algorithm is applied to layout a given graph automatically, the main desire of a user is to get a “nice” and stable configuration of the graph in a short time. In fact, this desire emphasizes two fundamental and important requirements:

1. The algorithm must be stable.

2. The algorithm must be quick.

In some cases, the Spring Modeling Algorithm described in Section 2.3.3 cannot fulfill these two requirements.

3.1.2 Trade-off between Speed and Vibrations

From the definition of spring forces and repulsion forces in Section 2.3.1, we know that there are two constant parameters C_s and C_r which will influence the result of computation of spring forces and repulsion forces directly. This influence will result in different values of displacement and movement for nodes in the graph and will ultimately determine the speed of layout.

Apparently, the bigger parameters C_s and C_r are, the higher the speed of automatic graph layout is. In fact, through experiments, we obtained the same results. We hope to increase the speed of layout by giving two parameters C_s and C_r large enough to speed up the procedure of layout. Unfortunately, this idea succeeds only occasionally. In most cases, this will result in a new problem appearing that is the *vibrations phenomenon*. We will describe the vibration phenomenon and the relationship between parameters and vibration phenomenon in detail in Section 3.3.1. Here we will explain it briefly. Vibration of nodes is a phenomenon that appears in the process of layout. Under the action of resultant forces, nodes in the graph should move closer and closer towards the stable position and the graph should change its configuration closer to the equilibrium configuration according to the algorithm. In fact, in some cases, because of the vibration, the nodes look like a reciprocator. This leads to a situation where the automatic graph layout cannot be finished successfully. In other words, the vibration phenomenon of nodes will influence the stability of the Spring Modeling Algorithm. Through experiments, we found that vibration appears not only when the values of parameters C_s and C_r are too big, but also when they are too small.

Therefore, the vibration phenomenon of nodes restricts the application of the Spring Modeling Algorithm. Because the reason of appearance of vibration phenomenon is that parameters C_s and C_r are too big or too small, the problem of vibration phenomena is not only a drawback of the Spring Modeling Algorithm; essentially, it is the drawback of the Spring Model.

When using the Spring Modeling Algorithm, in fact, user can avoid the vibration phenomenon by giving appropriate initialization settings and stop conditions. But in consideration of the speed, apparently the vibration phenomena and the speed of layout is a trade-off in the Spring Model. In a sense, if this trade-off can be solved, the two requirements mentioned in Section 3.1.1 can also be fulfilled.

Aiming at solving the trade-off between appearance of vibration phenomenon and the speed of automatic layout in the Spring Model, in the next chapter, we give our approach: the Dynamic Parameter Spring Modeling Algorithm, a modified spring modeling algorithm in which the basic aesthetic criteria existing in Spring Model mentioned in Section 3.1 continue to be maintained.

3.2 Dynamic Parameter Spring Model

3.2.1 Expected Position of Node

After the process of layout is finished, every node in the graph will move to a stable position called *final stable position*. If we know the final stable position for every node or a position close to the final stable position before laying out the graph, we can quickly move every node towards that position. However, before the layout, the user has no means to tell where the final stable position for every node will be.

According to the properties of spring, after layout, the distance between each pair of nodes connected by a spring will be close to the length of spring. In a graph, if two nodes are not connected directly, but the shortest path between them can be found, we can imagine that after layout, the real distance between these two nodes will be close to the shortest path multiplied by the spring length.

Therefore, the concept of *expected position* of a node can be used in the algorithm to speed up the process of layout. In the physical model, we can provide a mechanism that drives every node towards its expected position, and thus increase the speed of layout.

We make some experiments to test the validity of assuming the expected position of a node. We are more concerned with the validity of the assumption rather than its rationality. Through experiments, we found that by using the expected position, the algorithm is more effective for various types of undirected graphs.

In general, for a given graph, the assumptions for the majority of nodes are fairly precise. Only when large numbers of connection loops exist in a connected graph, our assumptions are not precise enough. However this lack of precision does not influence the stability of the algorithm; it has an indirect, slight influence on the final layout. This has also been proved by our experiments.

3.2.2 Dynamic Parameters

After the discussion based on the expected position of a node, we apply the concept of expected position to the new spring model. In the Spring Model, all the parameters are constant parameters, and this leads to the trade-off we mentioned before. If we want to solve the trade-off, we have to adjust these parameters. Therefore, in our approach, we redefine all the parameters and apply these new definitions of parameters to solve the trade-off.

At first, let us make an analysis of all parameters in the Spring Model at first. In Section 2.3, the definitions of forces are given in formulas (2.1) and (2.2).

$$F_s = C_s \log(d / C_d) \quad \text{.....(2.1)}$$

$$F_r = C_r / d^2 \quad \text{.....(2.2)}$$

According to the definition of spring forces and repulsion forces in the Spring Model, C_s determines the spring forces and C_r determines the repulsion forces. C_d represents the natural length of spring used in the Spring Model. The user needs to define these parameters before the layout. How to define these fixed parameters is still decided by the structure of graph and what types of layout the user wants to make. For example, the user can put a large graph in a small space by giving a small parameter C_d to increase the density; At the same time, he also needs to think about the consistency among these three parameters. Therefore the user finds it difficult to set the parameters in the Spring Model.

In general, parameters C_s and C_r represent a pair of related and proportional parameters in terms of the balance between spring forces and repulsion forces acting on every node in the system. In our approach, we give a similar definition.

In Section 3.2.1, we discussed how different nodes have different relative positions in a given graph and how these relative positions can be calculated by the concept of shortest path. Our proposal is to give every node in the graph a group of independent and irrelative parameters, which are called dynamic parameters. Thus we can adjust every parameter of the node in terms of the relative position in the graph and obtain the automatic layout in a short interval of time.

To be specific, we give each node i a group of independent parameters $C_s(i)$ and $C_r(i)$ with corresponding definitions for C_s and C_r in the original Spring Model. The variable i in $C_s(i)$ and $C_r(i)$ expresses that these two private parameters are related only to node i. This is the difference between the definitions of parameters in our approach and that in the original Spring Model where parameters have the same value for all nodes and they are constant throughout the process of layout.

3.2.3 Definition of Force Model

Figure 3.1 shows the Dynamic Parameter Spring Model in three-dimensional space. Due to the fact that graphs in real applications have become more complex, many of them are drawn in 3D space, not only in 2D space. Therefore we give the improved spring model in three-dimensional space. The model can be used in automatic graph layout in two-dimensional space as well.

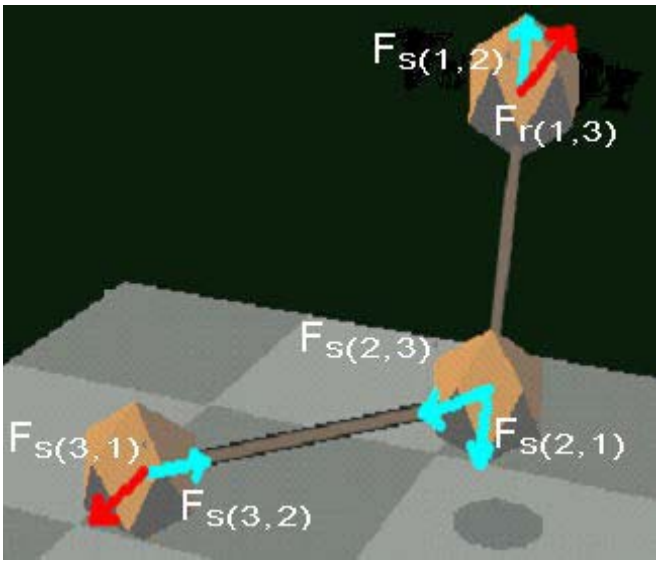


Figure 3.1 The Dynamic Parameter Spring Model

For every node i in the graph, the force acting on it should be calculated using the parameters of the node itself. For any node i in the graph, the spring force acting on it by the spring connected with node j is represented by $F_{s(i,j)}$. The repulsion force acting on the node i connected with node j is represented by $F_{r(i,j)}$. $C_s(i)$, $C_r(i)$ have the same meaning with C_s and C_r in the original Spring Model respectively. $C_s(i)$ and $C_r(i)$ are defined only for the node i, not for the whole system. The variable $d(i,j)$ is the real distance between node i and node j, and has the same definition as in the original Spring Model. Formulas (3.1) and (3.2) give the definitions of spring force and repulsion forces, and (3.3) shows the formula for the resultant of forces.

$$F_{s(i,j)} = C_s(i)\log\left(\frac{d(i,j)}{C_d}\right) \quad \text{.....(3.1)}$$

$$F_{r(i,j)} = C_r(i)/d^2(i,j) \quad \text{.....(3.2)}$$

$$F(i) = \sum_{j=1\,(\text{i,j are connected})}^{n} F_{s(i,j)} + \sum_{j=1(i,j\ \text{are not connected})}^{n} F_{r(i,j)} \quad \text{.....(3.3)}$$

After defining the forces, our next concern is how to define the dynamic parameters and how to calculate them. In subsection 3.2.1, we discussed the expected position of nodes. We can use the expected position for every node to replace the final position and to define the dynamic parameters based on this assumption. Thus we can make every node move quickly towards the expected position to increase the speed of layout.

In terms of different positions for a given node, we consider that two different groups of formulas will be used to compute the dynamic parameters $C_s(i)$ and $C_r(i)$ of every node.

If the real distance between the *fixed node* and node i is larger than the distance between the fixed node and its expected position, for every node i, $C_s(i)$ and $C_r(i)$ will be

$$C_s(i) = K_s \frac{d(f,i)}{sp(f,i)C_d} \quad \text{.....(3.4)}$$

$$C_r(i) = K_r \frac{d(f,i)}{sp(f,i)C_d} \quad \text{.....(3.5)}$$

respectively, where K_s and K_r are constant parameters predefined in the initialization stage of program running. C_d is the common constant parameter of the system. $sp(f,i)$ and $d(f,i)$ represent the shortest path and distance between the fixed node and every node i respectively. The fixed node is a unique node that is chosen by the user from all nodes of the graph before the procedure of layout.

If the real distance between the fixed node and node i is smaller than the distance between the fixed node and the expected position of node i, $C_s(i)$ and $C_r(i)$ will be

$$C_s(i) = K \frac{sp(f,i)C_d}{d(f,i)} \quad \text{.....(3.6)}$$

$$C_r(i) = K_r \frac{sp(f,i)C_d}{d(f,i)} \quad \text{.....(3.7)}$$

For these two situations, we use the maximum of the two ratios to adjust these parameters and speed up the computation. Based on the definition of dynamic parameters, we give the general formulas (3.8) and (3.9) for the Dynamic Parameter Spring Model.

$$C_s(i) = \max\left(\frac{d(f,i)}{sp(f,i)C_d}, \frac{sp(f,i)C_d}{d(f,i)}\right)K_s \quad \text{.....(3.8)}$$

$$C_r(i) = \max\left(\frac{d(f,i)}{sp(f,i)C_d}, \frac{sp(f,i)C_d}{d(f,i)}\right)K_r \quad \text{.....(3.9)}$$

Figure 3.2 shows a simple example and we can use it to illustrate the defined formula. The dotted circles represent the expected stable position of node i and node j. Node i is further from the fixed node than the expected position of i. Therefore the parameter will be $C_s(i) = K_r \frac{d(f,i)}{sp(f,i)C_d}$. As for node j, it is closer to the fixed node than its expected position, and therefore we use the following formula: $C_s(j) = K_s \frac{sp(f,j)C_d}{d(f,j)}$.

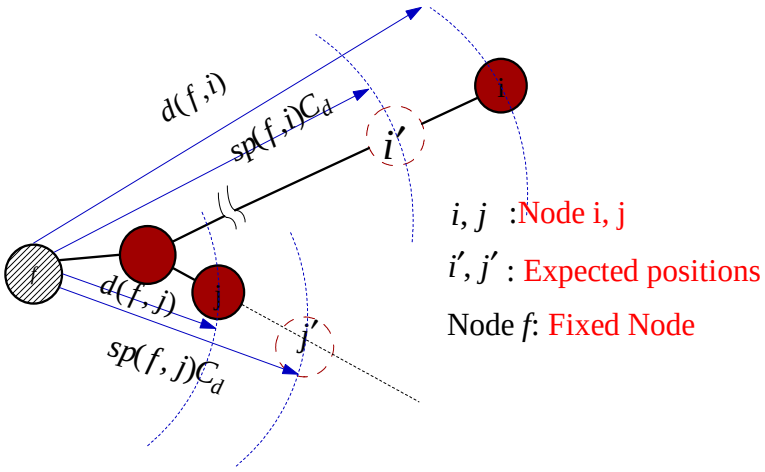


Figure 3.2 An example of DPSM graph

3.3 Dynamic Parameters and Avoiding Vibrations

3.3.1 Vibration Phenomena

Vibration phenomena can be found everywhere around us. Speaking of vibration, most people will imagine that two objects connected by a spring and these objects will make a rapid linear motion around an equilibrium position under the spring force. People take for granted that the vibration in the Spring Model is the same as the generic vibration we mentioned.

The truth is entirely different. The vibration phenomenon we mentioned in the Spring Model is a special state in the automatic graph layout. In the process of automatic layout, certain forces are computed and nodes are moved. The computations and movements are done several times without any interference from the user and each time a new layout is obtained until certain conditions are satisfied. In other words, the

automatic layout is not obtained in only one step, but in many steps and their number cannot be estimated. The description above corresponds to a successful automatic graph layout.

Sometimes the process of layout cannot be finished successfully. If we observe every step of layout, we will find that the same graphs appear after different layouts. This phenomenon gives the illusion that makes it look like an object (or a particle) appears in the same place many times. The same graph can appear after some consecutive layouts or not. In the Spring Model, if this type of phenomenon appears, in most cases we have consecutive layouts. The phenomenon makes it appear as some nodes are vibrating around some special equilibrium positions. We call this vibration phenomenon of nodes.

In the Spring Model, why does this type of phenomenon appear? The answer to this question can also give us a solution for the question of avoiding it in our DPSM.

Let us analyze a simple example.

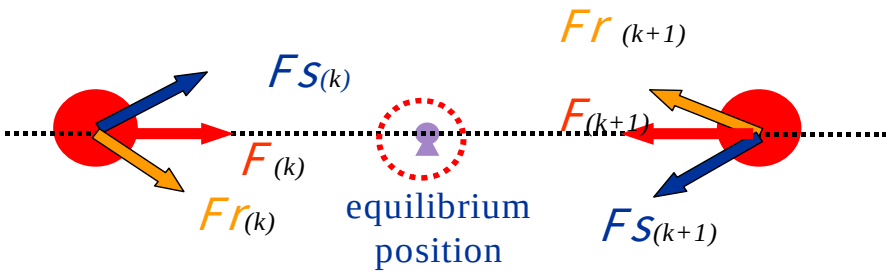


Figure 3.3 Vibration phenomenon

In this example (Figure 3.3), we observe a vibrating node in the graph. We assume that the left red circle represents the position after the $(k-1)^{th}$ layout. In the next layout, the resultant of force $F_{(k)}$ including both the spring force $F_{s(k)}$ and the repulsion force $F_{r(k)}$ will drive the node towards the next position, the position of the right red circle in Figure 3.3. After the k^{th} layout but before the $(k+1)^{th}$ layout, the node in the right position will move towards the left position under a similar force, Therefore the node will move between left position and right position and a stable state will never be reached, the process of automatic layout will never end.

3.3.2 Avoiding Vibrations in DPSM

Since we know why the vibration phenomenon appears, we should make this phenomenon disappear to ensure that the automatic graph layout can continue successfully. Because parameters in the original Spring Model need to be predefined, these parameters cannot be changed in the process of layout. Therefore, the force between two nodes is determined only by the distance between them.
In the DPSM, every node has a group of private parameters. In the process of layout, these parameters can be changed to be suitable for the status of the graph at that time. For a node far away from its expected position, private parameters will be assigned a bigger value to make the acting force bigger. Thus every node tends to move to its expected position.
The dynamic parameters also stop all nodes from moving away from their expected position. In real experiments, the dynamic parameters are effective in decreasing the possibility of appearance of vibration phenomenon.

In order to remove the influence of vibration phenomena entirely, another detecting algorithm has been added to the DPSM. Considering that the influence to the whole graph caused by each node is different, we only search the node with the biggest displacement in the graph and judge whether the node is in vibration or not to do the detection. After the detection, we examine 3 consecutive steps of layout. We consider the node vibrating if after the third step of layout, the node moves back to a place close to the position corresponding to the first step of layout. If a node is found in vibration, we need to:

1) set the position of the node after the second layout as the position for next step of layout *and*

2) decrease the constant parameters K_s and K_r to decrease the probability of appearance of vibration in subsequent layouts.

Figure 3.4 shows the detection of vibrations. If the distance $S_{(k-1,k+1)}$ is much smaller than the distance $S_{(k-1,k)}$ or $S_{(k,k+1)}$ which correspond to two consecutive layouts, we think there must be a vibration.

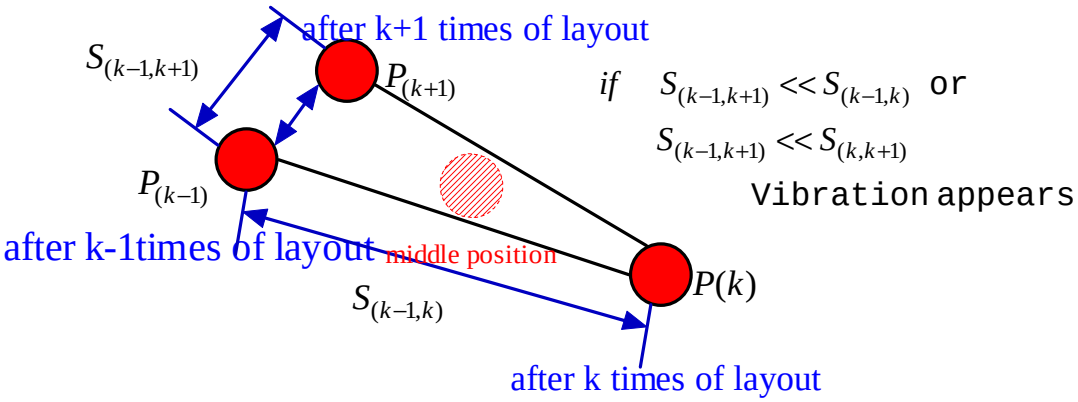


Figure 3.4 Vibration phenomenon detection

As far as vibrations of nodes are concerned, the important thing is to decrease the influence of vibration and not to avoid them completely. This is exactly what we achieve in DPSM.

3.4 Algorithms

Based on the definition of DPSM, this section will give the corresponding algorithm, the Dynamic Parameter Spring Modeling Algorithm (DPSMA). Section 3.4.1 will introduce a traditional algorithm to compute the shortest path for a given graph. The algorithm for dynamic parameter computing and adjusting will be introduced in section 3.4.2. DPSMA is given in section 3.4.3

3.4.1 All Pairs of Shortest Path

The concept of shortest path means finding the shortest path in the graph from one node to another node. The algorithm will become different if the edges have different weights. Figure 3.5 shows a small example of shortest path. In this graph, if the weights of all edges are the same, the shortest path between node *a* and node *d* is equal to 2, and the shortest path between node *b* and node *h* is 3.

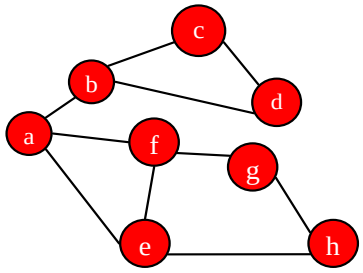


Figure 3.5 An example of shortest path

In DPSMA, for a given graph, all pairs of shortest path for the graph need to be computed before layout, and just once. Therefore the influence over the system of the computations of all pairs of shortest path is very small. As for the *DPSMA*, in order to compute all pairs of shortest path $sp(f,i)$ between the fixed node and any node i, we use the Floyd-Wars Hall algorithm [21]. The input to the algorithm is d_{ij}, which represents the distance between node i and node j (in the following code, *d[i][j]*) that is the shortest path between all pairs of nodes.

Algorithm:

Input adjacency matrix of graph with n nodes, W[0.. n-1][0..n-1]. Initialize shortest path matrix, *d[0..n][0..n];*

```
for i=0 to n (number of nodes)
{
   for j=0 to n
     d[i][j]=W[i][j];
}

for k=0 to n
{
  for i=0 to n
     for j=0 to n
       d[i][j] =min (d[i][j], d[i][k]+d[k][j]);
}
```

3.4.2 Dynamic Parameter Adjusting

The definitions of dynamic parameters $C_s(i)$ and $C_r(i)$ have been given in section 3.2.3. In the process of automatic graph layout, parameters need to be recomputed in every step of layout and readjusted to be suitable for the next steps.

Algorithm:

Compute all pairs of shortest path $sp(f,i)$ for every node;

Initialize const parameters K_s *and* K_r;

for i=0 to n (number of nodes)

{

compute $d(f,i)$;

if ($d(f,i) < sp(f,i)$) // two separate cases are needed

{

$C_s(i) = K_s d(f,i) / \left(sp(f,i) C_d \right)$;

$C_r(i) = K_r d(f,i) / \left(sp(f,i) C_d \right)$;

}

else

{

$C_s(i) = K_s \left(sp(f,i) C_d \right) / d(f,i)$;

$C_r(i) = K_r \left(sp(f,i) C_d \right) / d(f,i)$;

}

}

3.4.3 Dynamic Parameter Spring Modeling Algorithm

The Dynamic Parameter Spring Modeling Algorithm is based on the proposed Dynamic Parameter Spring Model that has been described in detail in Section 3.2. The following give the frame of the algorithm:

Algorithm:

Compute all pairs of shortest path $sp(f,i)$;
Initialize all positions of nodes;
Initialize all parameters;
Initialize stop condition;

While (graph does not reach stop condition)
{

for i=1 to n (number of nodes)

{

Initialize the $F(i)$

Compute dynamic parameters $C_s(i)$ and $C_r(i)$

for j=1 to n

{ compute the distance d;

if(node i and node j is is connected directly)

{ $F_{s(i,j)} = K_s C_s(i) \log\left(\frac{d(i,j)}{C_d}\right)$;
$F(i) = F(i) + F_{s(i,j)}$;
}
else
{ $F_{r(i,j)} = K_r C_r(i) / d^2(i,j)$;
$F(i) = F(i) + F_{r(i,j)}$;
}
}
compute displacement $\delta F(i)$ of every node i;

}
modify positions of all nodes;

if(vibration phenomena existing)
{
Adjusting parameter K_s and K_r;
Adjusting the positions of vibrating nodes;
}
compute the stop condition;
}

In the algorithm, in order to compute the distance d, we have two different formulas that are used in two-dimensional space and three-dimensional space respectively. In two-dimensional space, we use $d(i,j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ to compute the distance between node i and j. In three-dimensional space, the formula is $d(i,j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$. The constant parameters K_s and K_r are propositional parameters, and are used to balance the spring force and repulsion force. In DPSM, according to the user requirement of a "nice" layout and other aesthetic criteria, these two parameters need to be predefined in the initialization stage. It is possible that the algorithm in the process of layout can adjust these parameters automatically.

Chapter 4

Application to 3D-PP

4.1 Three Dimensional Pictorial Programming

4.1.1 Visual Programming System

Visual Programming System (VPS) provides the user with a new type of developing environment where pictorial objects are used to represent a paragraph of programs or applications. It is entirely different from the traditional programming technique in which the source is written in text form. In the visual programming environment, writing source code is replaced by managing some pictorial elements that make the programs friendly and easy to “write”. Expressions of the structure of programs by pictorial elements become more clear, intuitive and easy to understand.

4.1.2 Three Dimensional Visual Programming System

Based on two-dimensional visual programming systems, such as PP [22], ViewPP and others [23,24,25], 3D-PP [7,26,27] is our three-dimensional visual programming system that is extended from two-dimensional space. 3D-PP is the abbreviation of “Three Dimensional Pictorial Programming” which is one of our current research projects. Compared to two-dimensional VPS, 3D-PP can provide more pictorial objects or elements in the same limited area of display. That is also the essential difference between the three-dimensional visual programming systems and those in two-dimensional space.

3D-PP is designed for the parallel logic programming language GHC [28] which is one of the high level declarative programming languages. On one hand, a declarative programming language is suitable for visualization because visual programming is also declarative. On the other hand, a logic programming language requires comparatively fewer numbers of programming elements than a procedural language. Therefore, GHC is more suitable to be used in a visual programming language.

The clause of GHC is structured as follows:

predicates(arguments, …) : - guard | body.

In GHC, the basic program elements include atoms, lists, input data, output data, goals and built-in goals. Two or more goals can be contained in both body and guard. 3D-PP is built by using some pictorial programming elements. These pictorial programming elements are only the basic elements defined in GHC and expressed by some types of three-dimensional objects. The combination of these pictorial programming elements can be structured to express clauses in a GHC program as a graph. Programming is done to create or edit a graph by manipulating various types of three-dimensional pictorial programming elements in 3D-PP. These manipulations of 3D objects are equivalent to editing source codes in traditional programming systems. In other words, a graph in 3D-PP represents the visualization of a program and these 3D objects are just the visualization of basic programming elements of the program.

4.2 Implementation of DPSMA in 3D-PP

In 3D-PP, programs are expressed as graphs. Making a program has been transformed into manipulating a graph. From the point of view of programmers, the VPS of 3D-PP make the programming and editing programs more easy and intuitive. With the editing, modifying of graph, the positions of nodes represent the pictorial programming elements in the graph and the relationships among these nodes have to be changed accordingly. Thus after editing or modifying the program, both the nodes and the graph need to be redrawn and the relationships among the graph need to be maintained. The process of redrawing, in fact, is just a process of automatic graph layout.

The Spring Modeling Algorithm has been applied in 3D-PP to obtain the graph layout automatically. But for the problems mentioned, in many cases the Spring Modeling Algorithm restricts the visualization of programs in 3D-PP. Therefore in 3D-PP we use the DPSMA to replace the Spring Modeling Algorithm to make the automatic graph layout.

In 3D-PP, all pictorial elements are treated comparably although these pictorial elements represent different basic program elements in GHC clauses. The DPSMA has been given in Section 3.4.3, and we have pointed out that there are four requirements in the initializations steps:

A) Compute all pairs of shortest path $sp(f,i)$;

B) Initialize all positions of nodes;

C) Initialize all parameters;

D) Initialize stop conditions.

In 3D-PP, all the edges are regarded as springs with the same physical characteristics and all edges are supposed to have same weight. Therefore, to fulfill the first requirement (A) in the initialization steps of DPSMA, we compute all pairs of shortest path by the algorithm with the same weights. The algorithm has been given in Section 3.4.1. To fulfill the requirement (B), the initializations of all positions of nodes, we use the real values of coordinates of every node in the three-dimension orthogonal coordinates reference frame. To fulfill the requirement (C), the initializations of all parameters, we use different values in different cases. C_d and constant parameters K_s and K_r are not fixed in 3D-PP, but there are already some basic rules that can be used to define these parameters. For example, we usually use a smaller C_d for a complicated graph because we need to put many nodes in a limited space. In addition, K_s and K_r must be a pair of proportional parameters. To fulfill the requirement (D), the initialization of stop conditions, we define a constant, *force threshold,* as the stop condition. When the biggest resultant force among all nodes in the graph becomes smaller than the predefined force threshold, we think that the layout has reached its stable state and the computations should stop.

In addition, the fixed node is unique among all nodes and is selected by the user. In 3D-PP, the user can use the mouse to specify the fixed node by moving the mouse over a node.

By the application of DPSMA in 3D-PP, the graphs (GHC programs) become meaningful and understandable. Programmers can create, edit and modify the programs more easily. The vibrations phenomena when using the Spring Modeling Algorithm no longer appear. As far as the VPS of 3D-PP is concerned, the speed of automatic layout becomes much quicker than that of the Spring Modeling Algorithm.

Chapter 5

Application and Performance Evaluation

5.1 Properties of DPSM and Applications

5.1.1 Properties of DPSM

Because DPSM is a modified, improved spring model that is a best-known Force-Directed method, DPSM is also a Force-directed method. Therefore some common properties for Force-Directed Methods are all contained in DPSMA. Essentially, DPSM is still a force system defined by the vertices and edges, which provides a physical model for the graph. DPSMA is an algorithm based on a technique for finding the equilibrium state of the force system, that is, a position for each vertex, such that the total force on every vertex is zero [3]. An important property is the aesthetic criteria of automatic graph layout. Four basic aesthetic criteria still exist in DPSM: 1) Ensuring that edge lengths are uniform, 2) Minimizing edge crossings, 3) Revealing symmetry and 4) Distributing vertices evenly.

These same or similar properties decide that the DPSM and DPSMA have the same or similar applications. On one hand, the DPSMA is a method of generating an automatic graph layout. On the other hand, it represents a method of graph drawing. Users can get the result of layout for a given graph and can also observe the process of the automatic graph layout. On top of these common properties, DPSMA has some properties of itself.

3) Stability.

4) High speed

In DPSMA, we solve the problem of appearance of vibration phenomena. This ensures the stability of DPSM algorithm. Using dynamic parameters speeds up the process of layout and that makes the DPSMA a rapid algorithm.

5.1.2 Applications and Examples

The properties of DPSM show that the DPSMA can be largely applied in many fields. It can be used not only in scientific research but also in visualization of information.

To demonstrate the use of DPSMA, we show some steps in laying out a pentagonal shape in Figure 5.1. The process of layout is illustrated in Figures 5.1(1) to (6).

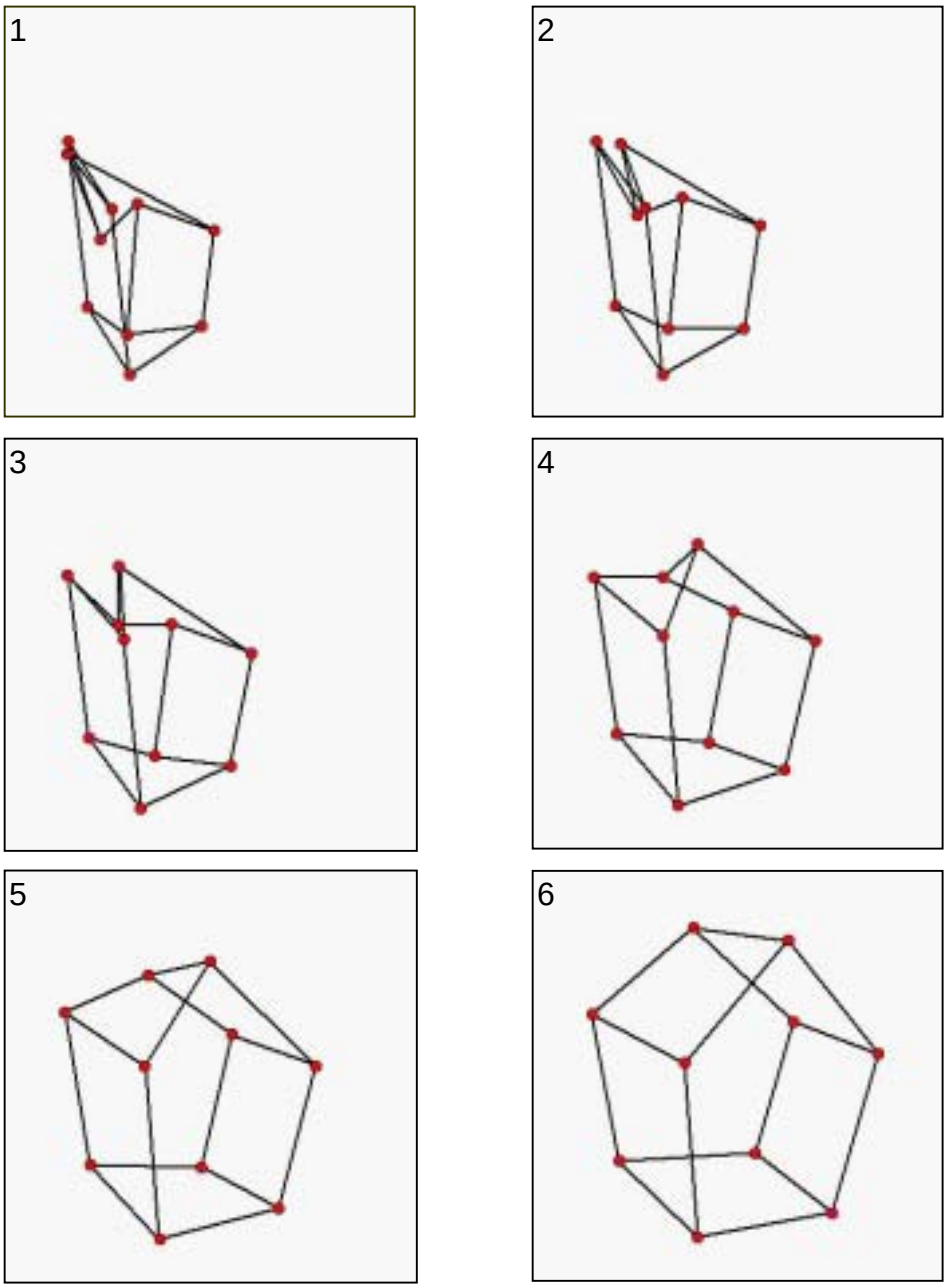


Figure 5.1 An example of laying out a graph automatically

For different types of undirected graphs, some examples are given to show the results of applications of DPSMA as follows (Figure 5.2). The most natural and aesthetic results from DPSMA are those obtained from symmetric graphs. Figure 5.2 illustrates some planar symmetric graphs.

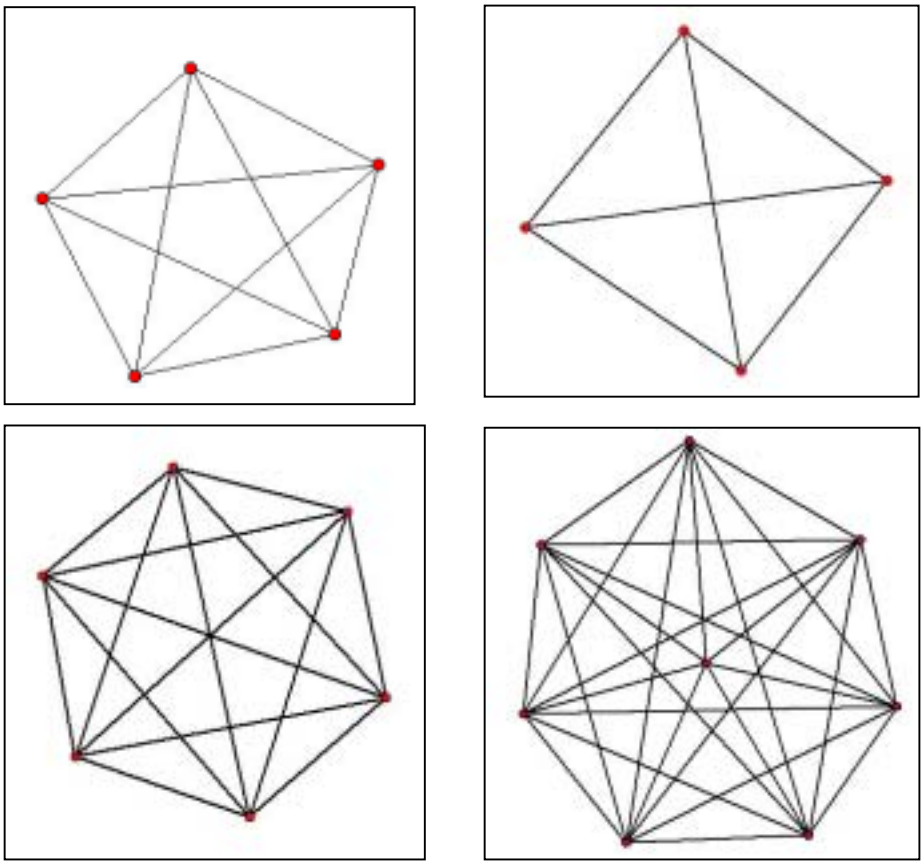
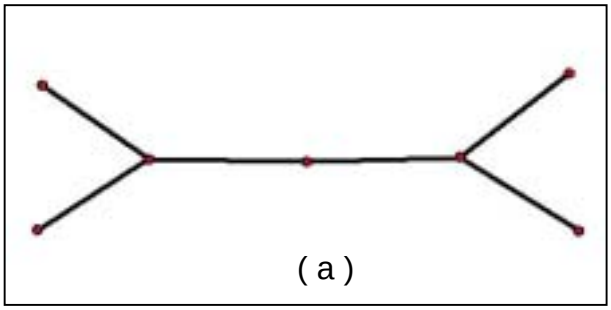


Figure 5.2 Layout of symmetric planar graph

Some examples of trees are represented by the DPSMA. Figure 5.3(a) shows a simple binary tree where Figure 5.3(b) illustrates a complicated binary tree.



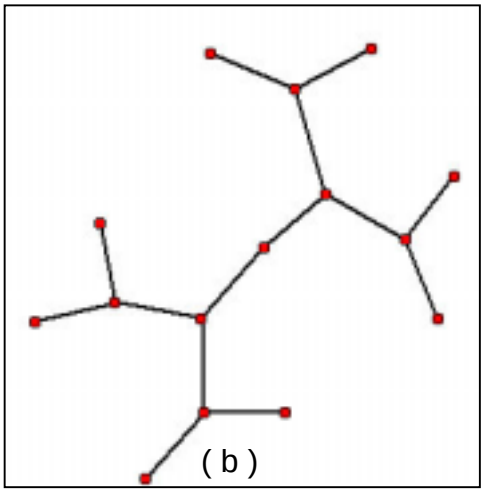


Figure 5.3 Representations of trees

We present in the following some examples of cubes. The layouts of these cubes are generated by the two-dimensional version of DPSMA and appear to be three-dimensional when laid out. In Figure 5.4, (a) represents one cube and (b) represents twin-cubes.

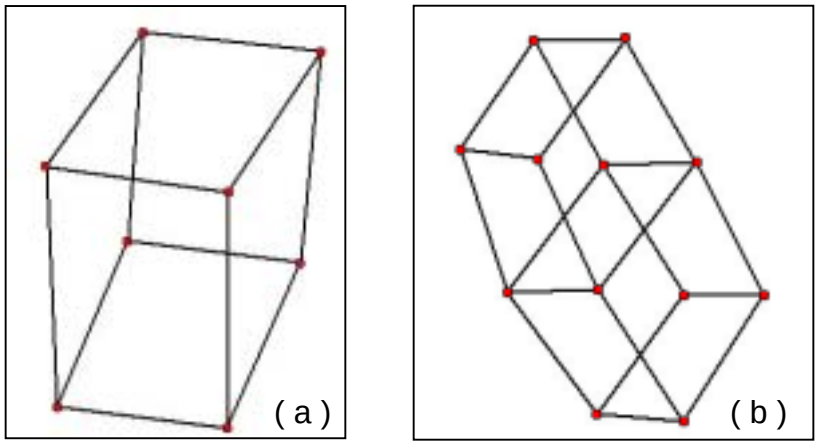


Figure 5.4 Cube and twin-cubes are laid out by the two-dimensional version of DPSMA

5.2 Performance Evaluations

We made some experiments to evaluate the performance of DPSMA. Because the improvements of the algorithm focus on its speed and stability (the problem of vibration phenomenon), these experiments are designed to evaluate the DPSMA in two respects:

1) Evaluate the speed of algorithms;

2) Determine whether or not vibrations phenomena appear.

For the first evaluation object, we use 5 undirected graphs with different complexity to compare the speed of layout between DPSMA and SMA. Considering the property of automatic graph layout, the comparisons include the comparison of speed in real running time and the comparison of number of layout iterations.

We made the same initialization settings for DPSMA and SMA to make the experiments. The force thresholds are defined as stop conditions of automatic layout. We defined a constant as the force threshold for a given graph in the initialization stage. When the biggest resultant forces among all nodes in the graph become smaller than the predefined force threshold, it means that the layout has reached its stable state and the computation should be stopped. This is the stop condition of the layout for a given graph.

In experiments, the force thresholds are set to 0.01(Newton), 0.05,0.1 and 0.5 respectively to evaluate the performance of DPSMA under different force threshold conditions. The parameters K_s and K_r are set to be the same with the initialization parameters C_s and C_r in the SMA respectively. Table 5.1 shows the basic data of 5 graphs in experiments.

Graph	1	2	3	4	5
Nodes	3	8	27	64	125
Edges	3	12	54	142	294

Table 5.1 Graph data

At first, we compare the number of iterations. In the same environment, we obtained the following results in Table 5.2.

No. of graph	Force threshold	0.5	0.1	0.05	0.01
1	SMA	41	45	47	52
	DPSMA	29	34	36	41
	DPSMA/ SMA	70.7%	75.6%	76.6%	78.8%
2	SMA	45	64	72	93
	DPSMA	34	51	53	74
	DPSMA/ SMA	75.6%	79.7%	73.6%	79.6%
3	SMA	127	176	199	261
	DPSMA	89	149	169	222
	DPSMA/ SMA	70.1%	84.7%	84.9%	85.1%
4	SMA	247	336	384	507
	DPSMA	176	264	311	386
	DPSMA/ SMA	71.3%	78.6%	81.0%	76.1%
5	SMA	278	479	575	801
	DPSMA	193	367	432	637
	DPSMA/ SMA	69.4%	76.6%	75.1%	79.5%

Table 5.2 Comparison of number of iterations

Figure 5.5 shows the comparison of number of iterations in the case of 4 different force thresholds as stop conditions. The x-axis is the number of graph and the y-axis is number of the iterations.

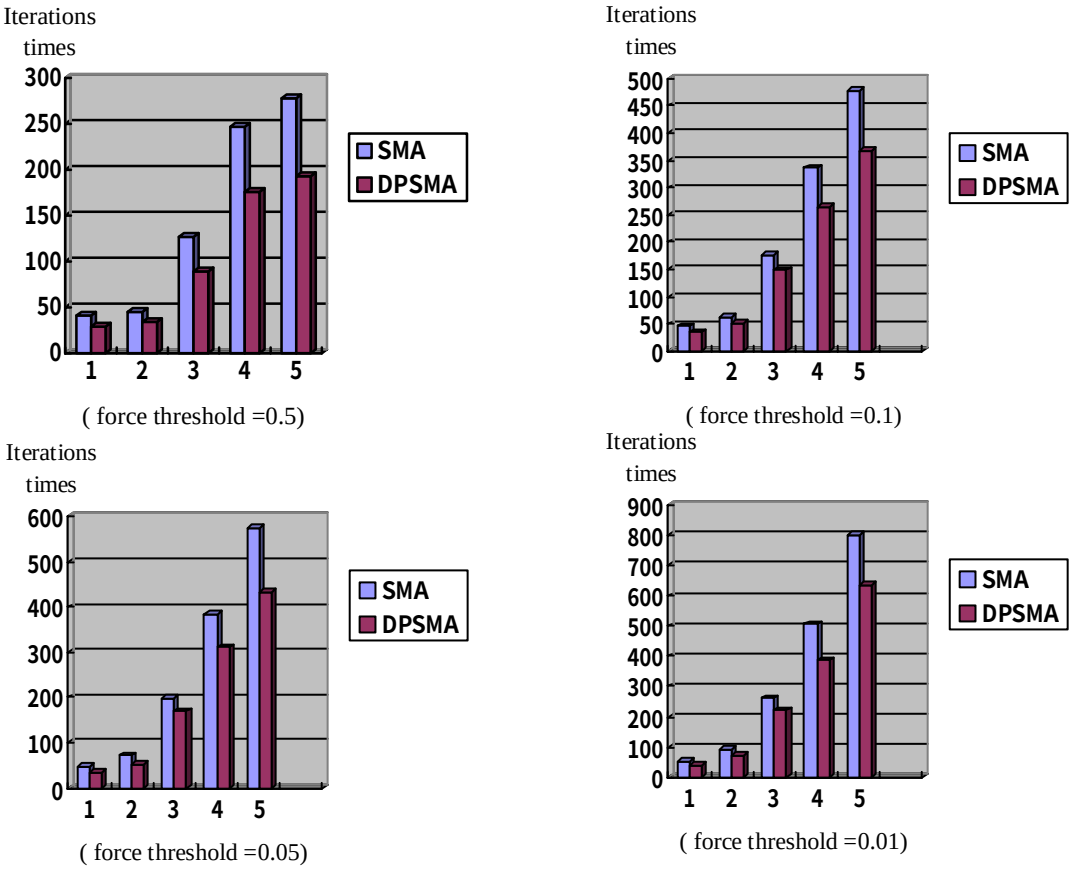


Figure 5.5 Comparison of number of iterations

We observe from Figure 5.5 that the number of iterations of automatic layout for a given graph by DPSMA is smaller than that obtained by SMA. The results also show that the DPSMA is more effective than SMA when the graph is complex, with many nodes and edges.

We also compare the real running time between DPSMA and SMA. In this experiment, the same 5 graphs are used to compare the performance. Because there are many factors that influence the real running time, such as the algorithm itself, optimizations of algorithm, real environment and others, we make the same experiments many times and use the average values as results.

Table 5.3 shows the results of experiments in the case of the 5 given graphs used previously.

No. of graph	Force threshold	0.5	0.1	0.05	0.01
1	SMA	118	125	141	138
	DPSMA	74	84	125	127
	DPSMA/SMA	62.7%	67.2%	88.7%	92.0%
2	SMA	132	177	229	335
	DPSMA	117	167	206	302
	DPSMA/SMA	88.6%	94.4%	90.0%	90.1%
3	SMA	179	216	267	367
	DPSMA	157	189	234	313
	DPSMA/SMA	87.7%	87.5%	87.6%	85.3%
4	SMA	3178	4407	4889	6139
	DPSMA	2834	3910	4319	5401
	DPSMA/SMA	89.2%	88.7%	88.3%	88.0%
5	SMA	38412	61340	71049	96324
	DPSMA	30132	43987	52987	71980
	DPSMA/SMA	78.4%	71.7%	74.6%	74.7%

Table 5.3 Comparison of real running time

(Unit: milliseconds.)

Figure 5.6 shows the comparisons of real running time in the case of different force thresholds as stop conditions for graph 4 and graph 5.

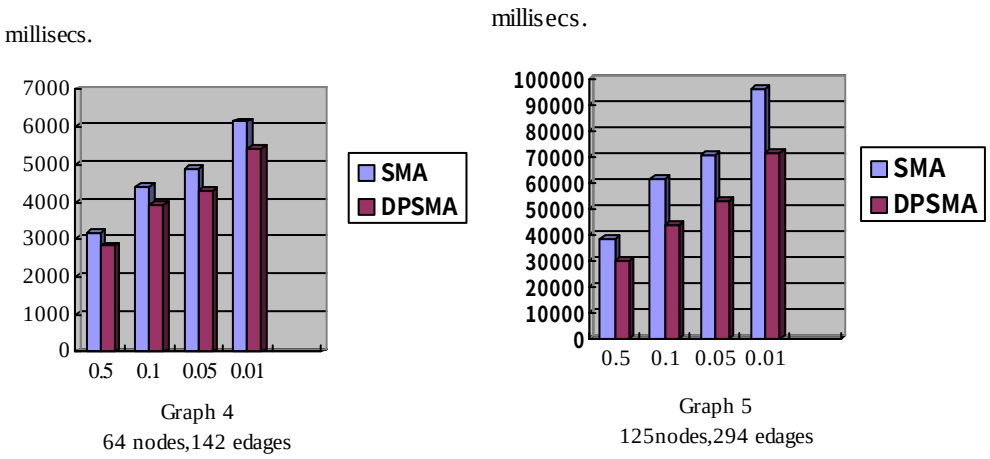


Figure 5.6 Comparison of real running time

From the comparison of speed in real running time, the DPSMA proves to be better than SMA. Because in DPSMA the computations of dynamic parameter and all-pairs of shortest path occupy real running time in every step of layout, the results are not so effective as in the comparison of number of iterations. The results also show that the DPSMA is more effective in the case of complex graphs.

These two groups of experiments show that the DPSMA is more effective than SMA with respect to both real running time and number of iterations.

We also made some experiments to evaluate the stability of DPSMA. No vibration phenomena are found in the experiments, and this shows that DPSMA is stable. When using the same data, in some cases the vibration phenomena are found in the process of layout in the case of SMA. This shows that SMA is not stable enough.

As for automatic layout for undirected graph, many experiments show that DPSMA is faster and more stable than SMA. For different types of undirected graphs, the efficiency of DPSMA is different. The efficiency in case of complex graphs is higher than that for simple graphs; the efficiency of low-precision graphs is higher than that of high-precision graphs.

Chapter 6

Conclusions

An improved spring model, the Dynamic Parameter Spring Model, which is applied in automatic graph layout for general undirected graphs has been proposed, and its corresponding algorithm, the Dynamic Parameter Spring Modeling Algorithm, has been implemented. The new improved model is based on the well-known Spring Model as the representation of force-directed approach to automatic graph layout. In the original Spring Model while trying to speed up the layout, the vibration phenomenon may appear and this can lead to an unstable algorithm.

Our model solves the existing problem in the original Spring Model and speeds up the process of automatic graph layout while avoiding the appearance of vibration phenomenon.

Using the new improved algorithm, a quick and stable automatic graph layout is provided and the user can understand the information in the given graph easily and intuitively. Our approach can also be used in various applications of information visualization.

Acknowledgements

I wish to have this opportunity to express my heartfelt thanks and profound gratitude to my supervisor Dr. Jiro Tanaka, Professor, University of Tsukuba, for his invaluable guidance, advice, supervision and constant encouragement during the course of the present study. It would not have been possible to complete the study without his generous training.

I am highly obliged to Dr. Buntarou Shizuki; Dr. Motoki Miura of Tsukuba University for their guidance and great deal of helpful advice for my research work.

I thank Mr. Tohru Ogawa; Mr. Kazuhisa Iizuka for their constructive and valuable advices and comments for my research work and all other members of VS group, Mr. Hideto Yamada; Mr. Okamura Toshiyuki; Mr. Masakazu Kanda for useful discussions for my research work.

I am grateful to all the reviewers who critically read my thesis and their comments improved the quality of this work.

Special thanks to Ms. Simona Mirela Vasilache; Mr. Iftikhar Azim Niaz; Ms. XiaoPing Ying, Mr. Hiroya Itoga and all other members of IPLAB, University of Tsukuba, for useful discussions, constructive criticism and timely help.

I wish to express my thanks to all my friends in China and Japan for their encouragement and helps.

Bibliography

[1] N. Quinn and M. Breur. A Force Directed Component Placement Procedure for Printed Circuit Boards. *IEEE Transactions of Circuits and Systems*, 26(6): 377-388, 1979.

[2] P. Eades. A Heuristic for Graph Drawing. *Congressus Numerantium* 42, 149-160, 1984.

[3] T. Fruchterman and E. Reingold. Graph Drawing by Force-Directed Placement. *Software – Practice and Experience* 21, 1129-1164, 1991.

[4] T. Kamada. Visualizing Abstract Objects and Relations, *World Scientific*, 1989.

[5] T. Kamada and S. Kawai. An Algorithm for Drawing General Undirected Graph. *Information Processing letters*, 31(1): 7-15, 1989.

[6] K. Sugiyama and K. Misue. Graph Drawing by Magnetic-Spring Model, Res. Rep. ISIS-RR-94-14E, Inst. *Social Information Science*, Fujitsu Labs Ltd., 1994.

[7] T. Miyashita and J. Tanaka. Integrating Three-dimensional Spring Model and Augmented Directed Manipulation technique. *Transactions of IPSJ*, 42(3): 565-576, 2001(in Japanese).

[8] C. Ware, G. Frank, M. Parkhi, T. Dudley. Layout for Visualizing Large Software Structures in 3D. *Proceedings of VISUAL'97*, 215-223, 1997.

[9] X. Liu, B. Shizuki and J. Tanaka. Dynamic Parameter Spring Modeling Algorithm for Graph Drawing, *Proceedings of the International Symposium on Future Software Technology (ISFST 2001)*, ZhengZhou, China, Nov. 5-8, 52-57, 2001.

[10] K. Sugiyama. Automatic graph drawing methods and their applications, *The Society of Instrument and Control Engineers*, 1993.

[11] C. Wetherell and A. Shannon. *Tidy Drawing of Trees. IEEE Transaction on Software Engineering*, SE-5(5): 514-520, 1979.

[12] E. Reingold and J. Tilford. Tidier Drawing of Trees, IEEE Transaction on Software Engineering, SE-7(2): 223-228, 1981.

[13] P. Eades, T.Lin, and X. Lin. Two Tree Drawing Conventions, *International Journal of Computational Geometry and Applications*, 3(2): 133-153, 1993.

[14] A. Lempel, S. Even and I. Cederbaum. An Algorithm for Planarity Testing of Graphs, *Theory of Graphs: Internae Symposium*, Rome, 215-232, 1967.

[15] G.D. Battista and R. Tamassia. Algorithms for Plane Representations of Acyclic Digraphs, *Theoretical Computer Science,* 61: 175-198, 1988.

[16] K.Sugiyama, S.Tagawa and M. Toda. Method for Visual Understanding of Hierarchical System Structures, *IEEE Trans. Syst. Man Cybern*, 11(2): 109-125, 1981.

[17] K.Sugiyama. A Cognitive Approach for Graph Drawing. Cyberrerics and Systems, 18(6): 447-488, 1987.

[18] A. Frick, A. Ludwig and H. Mehldau. A Fast Adaptive Layout Algorithm for Undirected Graphs. *Proceedings of the Symposium on Graph Drawing GD'95*, Springer-Verlag, 389-403, 1994.

[19] J. Nagumo and J. Tanaka. Introducing Fisheye-view into Graph Drawing Algorithm, *Transactions of IEICE*, 82-D-II(6): 1042-1048, 1999 (in Japanese).

[20] G.D. Battista, P. Eades, R. Tamassia and I.G. Tollis. Graph Drawing Algorithm for the Visualization of Graphs. *Prentice-Hall, Inc*, 1999.

[21] T.H. Corman., C.E. Leiserson. and R.L. Rivest, Introduction to Algorithm, *MIT Press*, 558-562, 1990.

[22] J. Tanaka. PP: Visual Programming System for Parallel Logic Programming Language GHC, *Parallel and Distributed Computing and Network'97*, 188-193, 1997.

[23] P.T. Cox, F.R. Glies and T. Pietrzykowski. Prograph: A Step towards Liberating Programming form Textual Conditioning, *IEEE Workshop on Visual Language, Rome*, 150-156, 1989.

[24] K. Kahm. ToolTalk – An Animated Programming Environment for Children, *Journal of Visual Languages and Computing*, 197-217, 1986.

[25] M. Toyoda, B. Shizuki, S. Takahashi, S. Matsuoka and E. shibayama. Supporting Design Patterns in a Visual Parallel Data-flow Programming Environment, *Proceeding of 1997 IEEE Symposium on Visual Language (VL'97)*, 1997.

[26] K. Miyagi, T. Oshiba and J. Tanaka. Three-Dimensional Visual Programming System 3D-PP, *15th Conference Proceedings Japan Society for Software Science and Technology*, 125-128, 1998 (in Japanese).

[27] T. Oshiba and J. Tanaka. "3D-PP": Three-Dimensional Visual Programming System, *Proceeding of 1999 IEEE Computer Society Press, Tokyo, Japan*, 1999.

[28] K. Ueda. Guarded Horn Clauses, *ICOT Technical Report*, TR-103, Institute for New Generation Computer Technology, 1985.