



Constrained Graph Layout

WEIQING HE

whe@cs.monash.edu.au

KIM MARRIOTT

marriott@cs.monash.edu.au

Department of Computer Science, Monash University, Clayton, Victoria 3168, Australia

Abstract. Most current graph layout technology does not lend itself to interactive applications such as animation or advanced user interfaces. We introduce the constrained graph layout model which is better suited for interactive applications. In this model, input to the layout module includes suggested positions for nodes and constraints over the node positions in the graph to be laid out. We describe four implementations of layout modules which are based on the constrained graph layout model. The first three implementations are for undirected graph layout while the fourth is for tree layout. The implementations use active set techniques to solve the layout. Our empirical evaluation shows that they are quite fast and give reasonable layout.

Keywords: constraints, graph layout, active set method, interaction, graphics

1. Introduction

Most research on graph layout has concentrated on how to layout a graph in some fixed style in isolation from the rest of the application. However, this model of graph layout, while very simple, does not lend itself to interactive applications such as animation or advanced user interfaces. This is for two main reasons.

The first reason is that, in many interactive applications, the graph is repeatedly modified (by either the user or the application program) and redisplayed. When the graph is redisplayed, the new layout should not be altered too much and so is able to preserve the *mental map* [14, 34], called *layout stability* in [38, 2], of the user, that is the new layout should not move an existing node unless the current position leads to poor layout. Consider Figure 1 for example. A user may initially draw a graph by hand, Figure 1(a), and obtain a satisfactory layout, Figure 1(b). The user then interactively adds one node and two edges to the existing graph to form a new graph, Figure 1(c), with the hope of drawing a graph with layout similar to that shown in Figure 1(d). However, a static graph layout algorithm such as Kamada's [28] algorithm will produce the layout, Figure 1(e), which makes the user totally lose his/her orientation.

The second reason is that almost all existing graph layout algorithms are quite restrictive in how graphs are laid out since they encapsulate fixed layout aesthetics. The application program cannot place constraints on the layout which express the underlying semantics of the object represented by the graph. Take telecommunication networks as an example [7]. A telecommunication network must interconnect every pair of users by means of communication links selected so as to make up a connected chain. If geographical distance and location are ignored and the only concern is with incidence relationships, then a graph, whose vertices represent terminal stations or switching centres and whose edges represent communication links, can be used to model a telecommunication network. Figure 2(a) gives a sample graph of telecommunication network. For the layout of a telecommuni-

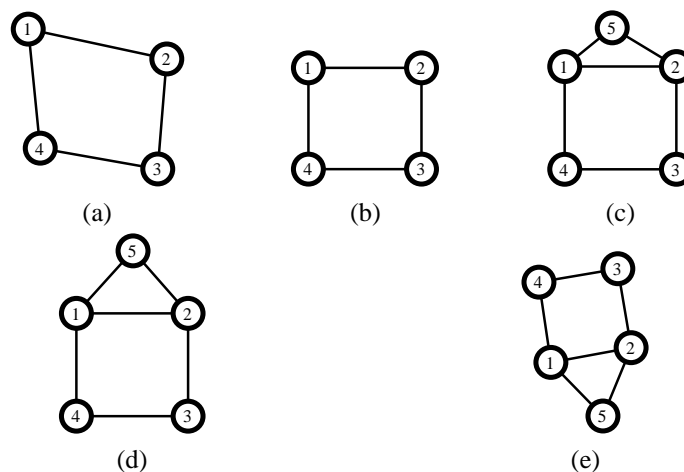


Figure 1. Example of mental map preservation.

cation network graph, it is natural to seek a layout, which is aesthetically pleasing, but in which vertices representing terminal stations are placed on the external boundary of a given rectangular display area because they have direct connections to subscribers, and vertices representing switching centres are placed internally since they usually have no direct connections to subscribers. If a graph of telecommunication network is so constrained, a possible layout of Figure 2(a) could look like that given in Figure 2(b).

To overcome these problems, we introduce a general model of *constrained graph layout* which is better suited for interactive applications involving graph layout, see Figure 3. In constrained graph layout, the graph layout module takes: the graph, a set of constraints over the x and y positions of the nodes, and a partial assignment of suggested values for the node coordinates. The graph layout module is responsible for finding an assignment to variables representing the node coordinates which is *feasible*, that is satisfies the constraints, gives a good layout, and assigns values to the variables which are as close as possible to the suggested values. The constraints enable the layout module to take additional semantic information about the graph into account, and the suggested values allow the layout module to try and preserve the current layout of the graph. Different graph layout modules allow different classes of constraints and may also embody different layout algorithms and aesthetic criteria. For different applications one must use the appropriate graph layout module.

Our major technical contribution is the implementation and design of four different constrained graph layout modules. All four modules allow arbitrary linear arithmetic constraints. The first three modules are for undirected connected graphs, while the fourth module is for tree layout. The first module is based on Kamada's aesthetic cost function [28, 29]. The second module is a modification of this cost function which removes the non-polynomial terms. And the third module is an integration of the first and the second ones. Unfortunately, Kamada's algorithm for graph layout is not suitable for constrained

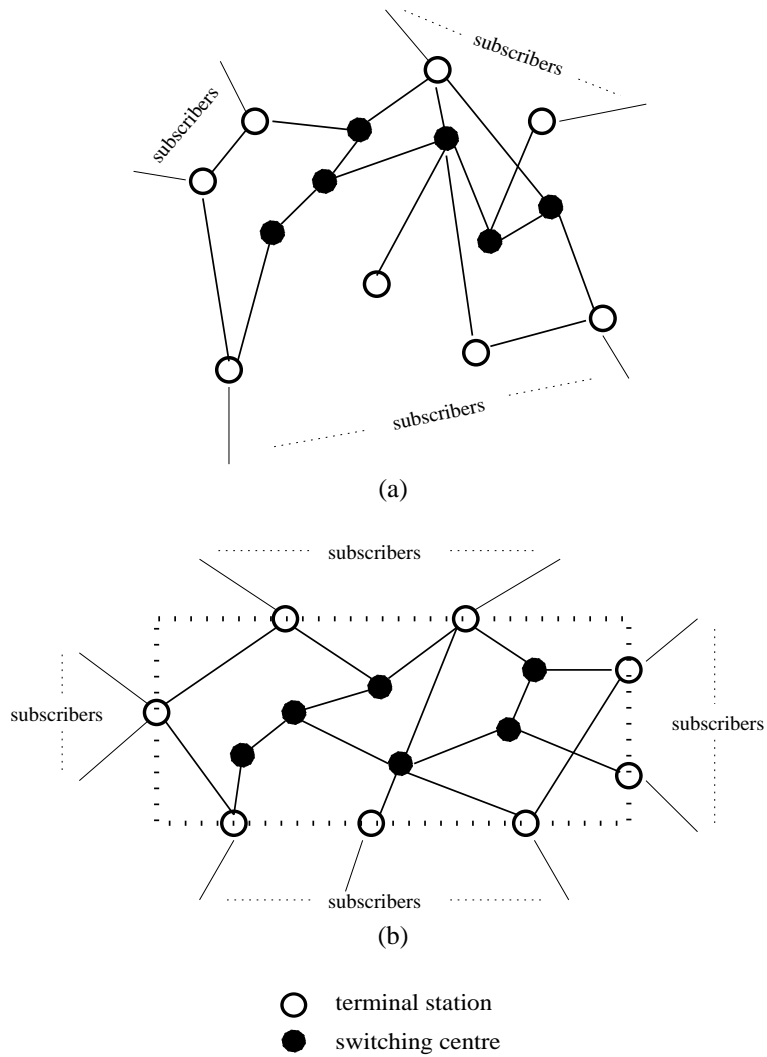


Figure 2. Example of constrained graph layout.

graph layout since it cannot handle interaction between variables. Instead we use an active set method developed to solve constrained optimization problems with an arbitrary non-linear objective function. The tree-layout module relies on methods developed to solve quadratic programming.

We also present an empirical evaluation of the four modules. Our evaluation shows that our second layout module for undirected connected graphs is not significantly slower

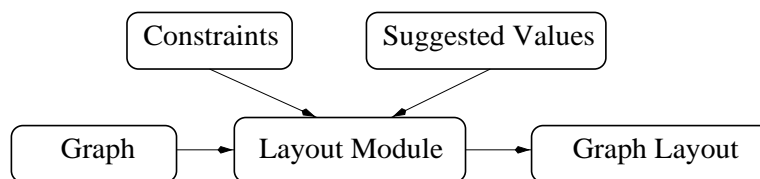


Figure 3. The constrained graph layout model.

than Kamada's original algorithm for unconstrained graph layout - thus there is very little performance penalty in using this more flexible model. Second, we show that the second module gives faster and more robust layout, but sometimes the layout is inferior to that given by the first model (which is based on Kamada's original cost aesthetic). Third, we show that the third module gives the same layout as the first model but is significantly faster. Fourth, we show that the tree layout module is much faster than the other two modules. Thus, if you are only laying out trees you should use this specialized module.

The remainder of this paper is as below: Section 2 briefly reviews related work. The general constrained graph layout model is introduced in Section 3. Section 4 and Section 5 detail the implementations of the general constrained graph layout model for the first three modules and the fourth module - the one for tree layout, respectively. Section 6 provides a detailed empirical evaluation of the different constrained graph layout modules. Finally, possible applications and future work are discussed in Section 7.

2. Related Work

Until recently there has been relatively little work on graph layout in the presence of both constraints and suggested values. The work most closely related is that of Kamps and Klein [30]. They also look at layout in the presence of constraints. The class of constraints they allow, called geometric constraints, is less expressive than the arbitrary linear arithmetic constraints considered here. For example, they cannot specify that a node should be placed at the mid-point of two other nodes. Furthermore, they do not allow suggested values for nodes.

Other closely related work is described in Lüders *et al* [32] who use a two-phase combinatorial optimization algorithm for graph layout. They also allow constraints, which are inequalities between node positions. Again, their constraints are not as expressive as those considered here.

Dengler *et al* [12] also look at constraint-driven layout. Their motivation is quite different. In their approach constraints are generated automatically and encode good graphic design rules. They do not require all of the constraints to be satisfied, rather the layout algorithm just tries to satisfy as many of these constraints as possible.

Kamada [28] presents an algorithm for drawing general undirected graphs; and the algorithm can be modified to constrained versions for either *radial drawings* by constraining the vertices to lie on concentric circles or *layered drawings* by assigning the hierarchy level

to each vertex. In addition, Kamada [28] also introduces a constrained-based object layout system – COOL. COOL includes a linear constraint solver which is able to solve constraints representing relations like order, average, horizontal, vertical and so on, and produces layout in two separate steps. COOL first sends the graph to be laid out to its general undirected graph drawing module, which outputs an initial layout without considering the constraints. The layout is in the form of linear constraints over the nodes' x and y coordinate variables. Next, the constraint solver of COOL collects these constraints and solves them together with the constraints generated by the application. Our approach is better because the two stages are combined into a single stage. This allows better layout since constraints influence the layout immediately. We also allow arbitrary linear constraints.

There is, of course, a considerable body of literature to do with constraint based generation of diagrams, in which layout is hardwired into the rules of generation. See, for example, Brandenburg [5], Helm and Marriott [25, 24], Weitzman and Wittenburg [47] and Cruz *et al* [10]. There is also a considerable body of literature on constraint based diagram manipulation. See for instance, Garnet [35], QOCA [26] and ThingLab [3].

More general related work is described in the survey [1]. In particular, the “spring” idea for drawing general graphs was introduced by Eades [13] and other force-directed graph drawing algorithms for unconstrained graph layout can be found in [45, 11, 28, 19, 41, 18]. [6] contains an experimental comparison between [45, 18, 11, 28, 19]. Linear programming and quadratic programming are used in some stages for directed graph drawing [16, 20, 42]. Algorithms for incremental unconstrained graph layout are given in [9, 36], while [31] discusses the approach of integration of declarative and algorithmic graph layout.

Mental map preservation of diagram was first proposed by Eades *et al* [14]. Misue *et al* [34] give a *force scan algorithm*, like a spring layout algorithm for graph layout, which performs diagram adjustment to push overlapping node apart, while keeping the mental map of a given layout. Paulisch *et al* [38] talked about a similar concept using the terminology “graph layout stability” and [2] uses constraints to achieve layout stability. Lyons [33] tries to improve the distribution of the nodes in the new layout according to certain measures of distribution (*cluster busting*) while simultaneously trying to minimize the difference between the two layouts according to measures of difference (*anchored graph drawing*). Papakostas *et al* [37] present scenario, which is conceptually very close to the *suggested values* in our layout model, called *relative-coordinates*. However that is for interactive orthogonal graph layout. Thus it only concerns the *mental map* from the view point of edge bends and drawing area. Storey *et al* [40] provide a layout adjustment algorithm which can be used to create fisheye views of nested graphs.

A preliminary version of this paper appeared in [23].

3. Constrained Graph Layout

Existing research on graph layout has tended to focus on how to layout a graph statically using a fixed pre-defined style. Unfortunately, this model of graph layout, while very simple, does not lend itself to interactive applications since when a graph is modified and redisplayed the new layout may not preserve the mental map of the user and the application program cannot add constraints on the layout which take into account the underlying semantics of the object represented by the graph.

For these reasons we have introduced the *constrained graph layout* model which overcomes these problems and so is better suited for interactive applications. In constrained graph layout, the graph layout module takes three parameters. The model is shown in Figure 3. The first parameter is a *graph*, $G = \langle V, E \rangle$, where $V = \{1, \dots, n\}$ is the set of nodes in the graph where each node is represented by a (unique) integer and E is a set of edges $\langle i, j \rangle \in E$ for $i, j \in V$. The second parameter is a set of *constraints* over the x and y position of the nodes, where x_i and y_i are variables denoting the x and y coordinate of node i in the layout. Note that the constraints may also refer to other variables than the node coordinates. The third parameter is an assignment, ψ , of *suggested values* for the variables representing the node coordinates. For example, $\psi(x_i)$ is the desired value for the x coordinate of node i . Each assignment to a variable, v , has an associated weight, $w(v)$, indicating the importance of the suggested value. The larger the weight, the more the value is desired. If the weight is zero, i.e. $w(v) = 0$, then the suggested value is ignored. In the constrained graph layout model, the constraints enable the layout module to take additional semantic information about the graph into account, and the suggested values allow the layout module to try and preserve the current layout of the graph.

The graph layout module is responsible for finding an assignment to variables representing the node coordinates which satisfies the constraints, gives a good layout, and assigns the values to the variables which are as close as possible to the suggested values. More exactly, the graph layout module embodies an algorithm to solve the optimization problem:

$$\text{minimize } \phi(\vec{v}) + \text{dist}(\psi(v), v) \text{ with respect to } C(\vec{v})$$

where $\vec{v} = (x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_n)$ and $\phi(\vec{v})$ captures the graph layout aesthetic as an objective function over the nodes' x coordinates and y coordinates, $C(\vec{v})$ is a set of constraints and dist , which takes into account the weight, is some metric over the variable values.

Different graph layout modules support different classes of constraints. For example, one layout module might accept arithmetic equalities while another might allow linear arithmetic equality and inequality constraints. Of course there is a trade off between the speed of layout and the expressiveness of the allowed constraints. Different graph layout modules may also embody different layout algorithms and aesthetic criteria. For example, a layout module might be specialized for tree layout in which the criterion is to minimize the size of the tree or else the module might be for general directed graph layout, in which the criteria for layout are to minimize the number of edge crossings and to represent isomorphic sub-graphs identically. The final way in which layout modules may differ, is in the choice of metric by which solutions are compared. For instance, the metric might be the Euclidean norm or it might be the function which gives 0 if the values are identical and 1 otherwise. Here we will use the weighted square of the Euclidean distance.

For different applications one must use the appropriate graph layout module. That is, one should choose a module which allows expressive enough constraints, yet is also efficient enough. In the next two sections we describe four different graph layout modules which we have implemented.

4. Implementation for Undirected Graphs

In this section we detail three constrained graph layout models for general undirected connected graphs. All three models handle arbitrary arithmetic linear equality and inequality constraints and use the square of the Euclidean distance as the metric over solution values with the contribution of each variable multiplied by the associated weight. The basic idea is to use a spring model energy function as the aesthetic cost function ϕ , however the models differ in the choice of ϕ .

The first model, *Model A*, uses the aesthetic cost function suggested by Kamada [28]. This is

$$\phi_A = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{1}{2} k_{ij} (|p_i - p_j| - l_{ij})^2 \tag{1}$$

where: k_{ij} is a spring factor between node p_i and node p_j ; and
 l_{ij} is a desirable length between node p_i and node p_j .

Intuitively, the model places a “spring” between each pair of nodes which tries to position the nodes so that the distance between them is the desired length. The cost function is a measure of the energy in the springs. Following [28], l_{ij} is defined by $l_{ij} = L \times d_{ij}$ where L is the desirable length of a single edge in the display plane and d_{ij} is the length of the shortest path(s) between node v_i and v_j and k_{ij} is defined by $k_{ij} = K/d_{ij}^2$ where K is a constant. We can rewrite the cost function (1) to the following:

$$\begin{aligned} \phi_A = \frac{1}{2} \sum_{i=1}^{n-1} \sum_{j=i+1}^n k_{ij} \{ & (x_i - x_j)^2 + (y_i - y_j)^2 \\ & + l_{ij}^2 - 2l_{ij} \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \}. \end{aligned} \tag{2}$$

Our second model, *Model B*, is a polynomial approximation of *Model A*.

$$\phi_B = \sum_{i=1}^{n-1} \sum_{j=i+1}^n k_{ij}^2 ((x_i - x_j)^2 + (y_i - y_j)^2 - l_{ij}^2)^2. \tag{3}$$

In this model the cost function is the sum of the squares of the squared differences between the desired distance between nodes and the squared actual distance between nodes. The definitions of k_{ij} and l_{ij} are the same as those in (2).

The primary disadvantage of *Model B* over *Model A* is the weaker repulsive force between nodes which arises from the lack of $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ like terms in the aesthetic cost function of *Model B*. This means that *Model B* may sometimes produce a layout with some coincident nodes. In the case of *Model A*, however, since ϕ_A has $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ like terms, nodes can never be assigned the same location since

$$\frac{\partial}{\partial x_i} \left(\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \right) = \frac{(x_i - x_j)}{\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}},$$

which is undefined when $x_i = x_j$ and $y_i = y_j$. In other words, ϕ_A will have some points where its partial derivatives do not exist, and so any local minimum search method will never go to these points.

On the other hand, lack of smoothness of the partial derivatives of ϕ_A means that the computation of the minimum of ϕ_A may be sensitive to the initial configuration and initial feasible solution. This is because the optimization method may not be capable of leaping over a point where the partial derivative does not exist from the current feasible solution to a solution closer to the local minimum. Since ϕ_B has no points whose partial derivatives do not exist, we would expect numerical optimization techniques to be more stable when solving *Model B*. This is of particular importance in the case of constrained optimization because the presence of constraints makes it more difficult to find an initial feasible solution which is close to the global minimum.

The other main advantage of *Model B* over *Model A* is that we would expect to be able to compute a minimum of ϕ_B faster than we can compute a minimum of ϕ_A . This is because one of the main time costs in computing a minimum in almost any numerical method is the need to calculate partial derivatives and computing partial derivatives of ϕ_A will take longer because of the $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ terms. In particular, for popular optimization techniques such as *gradient descent method* and *conjugate gradient descent method* [17], the increment in line search can be determined symbolically if the aesthetic cost function has polynomial form while only numerical methods can be used if the aesthetic cost function includes $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ terms.

Model A and *Model B* can be integrated into a two-phase procedure, called *Model C*, which gives the benefits of both models. In this model the aesthetic cost function is:

$$\phi_C = \begin{cases} \phi_B & \text{phase one,} \\ \phi_A & \text{phase two.} \end{cases} \quad (4)$$

The idea is, in phase one, to use *Model B* to quickly obtain a certain local minimum, then see whether node coincidence occurs or not, if not, layout terminates; otherwise, phase two starts. This is: shifts nodes that are coincident slightly along different directions to make every node a different point, then call *Model A* to finalize the layout. The intuition behind the model is that phase two should not take too long because some local minimum should be close, and the layout that phase two produces will have no node coincidence.

The most important question is how can we efficiently solve the resulting optimization problems when using any of the three models for constrained graph layout. In all cases we must solve a constrained optimization problem of the following form:

$$\begin{array}{ll} \text{minimize} & \phi^*(\vec{v}) \\ \text{subject to} & C(\vec{v}) \end{array} \quad (5)$$

where $\vec{v} = (x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_n)$, C is a set of linear equality and inequality constraints over \vec{v} , and

$$\phi^*(\vec{v}) = \begin{cases} \phi_A(\vec{v}) + \sum_{v \in \vec{v}} w(v) \cdot (\psi(v) - v)^2 & \text{for Model A;} \\ \phi_B(\vec{v}) + \sum_{v \in \vec{v}} w(v) \cdot (\psi(v) - v)^2 & \text{for Model B;} \\ \phi_C(\vec{v}) + \sum_{v \in \vec{v}} w(v) \cdot (\psi(v) - v)^2 & \text{for Model C.} \end{cases}$$

Kamada [28] gives a simple and efficient algorithm to minimize ϕ_A in the case that there are no constraints. He claims that the $2n$ -dimensional Newton-Raphson method cannot be directly applied because $\sum_1^n \frac{\partial \phi_A}{\partial x_i} = 0$, and $\sum_1^n \frac{\partial \phi_A}{\partial y_i} = 0$, which means the $2n$ partial derivatives are not independent of one another. Instead his algorithm repeatedly recomputes the position of each node, one at a time, by solving two linear equations, involving calculation of derivatives only for that one node, to obtain x - and y -increments of the node to be moved while the other nodes are temporarily frozen. The algorithm terminates when a local minimum is reached. Unfortunately, it seems impossible to use Kamada's algorithm to solve problems of the form of (5) because the constraints introduce interaction between variables and so it is not possible to recompute the position of a node independently from the current position of the other nodes. Therefore we need some other method for solving optimization problems of the form of Equation (5).

Our implementations for *Model A*, *Model B* and also *Model C* are instead based on the *Active Set Method* [17]. This is an iterative technique developed to solve constrained optimization problems with inequality constraints. It is reasonably robust and quite fast. The key idea behind the algorithm is to solve a sequence of constrained optimization problems O_1, \dots, O_n , which only have equality constraints. This set of equality constraints, \mathcal{A} , is called the *active set*. It consists of the original equality constraints plus those inequality constraints which are required to be equalities. The other inequalities are ignored.

Essentially, each optimization problem O_i is solved using a *gradient descent method* which iteratively computes a solution which is feasible with respect to the equality constraints in O_i and which is in a search direction \vec{d} reducing the objective function. However, it may be that the new solution while feasible with respect to the active set of O_i , holds $\vec{v} \cdot \vec{a} = b$ for an inequality constraint $\vec{v} \cdot \vec{a} \geq b$ of the original problem which is not in the active set. In this case the corresponding equality $\vec{v} \cdot \vec{a} = b$ is added to the active set, giving rise to a new optimization problem O_{i+1} . Constraints may also be taken out of the active set, when a better search direction can be found "away" from the constraint in the direction satisfying the original inequality.

The precise algorithm is given in Figure 4. Assume that $\mathbf{C} = \{c_1, c_2, \dots, c_m\}$ and that $\mathbf{a} = (\vec{a}_1, \vec{a}_2, \dots, \vec{a}_m)$ and $\mathbf{b} = (b_1, b_2, \dots, b_m)$ where \vec{a}_i and b_i are the coefficient column vector and right-hand side constant of c_i , respectively. At each step $\mathcal{A} = \{\vec{a}_i : c_i \text{ is active}\}$ is a column vector set representing the active set.

The algorithm in Figure 4 proceeds as follows. In step (b), an initial feasible solution is found, and the initial active set \mathcal{A} is set to $\mathcal{A}(\vec{v}^{(1)})$, which is the set of \vec{a}_i such that $\vec{v}^{(1)} \cdot \vec{a}_i = b_i$ holds. Steps (c) to (h) are then performed iteratively. In each iteration $\vec{v}^{(k)}$ is a feasible point and step (c) attempts to solve the optimization problem

$$\begin{aligned} & \text{minimize} && \phi^*(\vec{v}^{(k)} + \vec{d}) \\ & \text{subject to} && \vec{d} \cdot \vec{a}_i = 0, \vec{a}_i \in \mathcal{A}. \end{aligned} \tag{6}$$

The solution to this problem is the search direction \vec{d} . If \vec{d} is small enough, that means $\vec{v}^{(k)}$ is an acceptable solution, and the Lagrange multipliers $\vec{\lambda}^{(k)}$ of the problem are examined, in step (d), to determine if a local minimum has been reached.¹ Denote $\lambda_q^{(k)} = \min \lambda_i^{(k)}$ for all $\vec{a}_i \in \mathcal{A}$ and c_i is an inequality constraint. The algorithm terminates with $\vec{v}^{(k)}$ as the solution if $\lambda_q^{(k)}$ is non-negative because if all λ_i are non-negative there exists no better solution near $\vec{v}^{(k)}$, and so $\vec{v}^{(k)}$ is a local minimum. Otherwise c_q should be removed from the active set,

```

(a) Compute  $l_{ij}, k_{ij}$  for  $1 \leq i \neq j \leq n$ ;
(b) Compute initial feasible solution  $\vec{v}^{(1)}$  and active set  $\mathcal{A} := \mathcal{A}(\vec{v}^{(1)})$ ;
     $\phi_{old}^* = \phi^*(\vec{v}^{(1)})$ ;
(c) Compute  $\vec{d}$  by solving (6);
    if ( $\vec{d} \geq \epsilon$ )
        goto (e);
(d) Let  $\lambda_q^{(k)}$  solve  $\min \lambda_i^{(k)}$ , for all  $\vec{a}_i \in \mathcal{A}$  and  $c_i$  is an inequality constraint;
    if ( $\lambda_q^{(k)} \geq 0$ )
        terminate with  $\vec{v}^{(k)}$  as the solution;
    else
        remove  $\vec{a}_q$  from  $\mathcal{A}$ ; //This is to remove  $c_q$  from the active set.
        goto (c);
(e) Compute  $\bar{\alpha}^{(k)} := \min \frac{b_i - \vec{v}^{(k)} \cdot \vec{a}_i}{\vec{d} \cdot \vec{a}_i}$  for  $\vec{a}_i$  in  $\bar{\mathcal{A}}$  and  $\vec{d} \cdot \vec{a}_i < 0$ , and
    choose  $\alpha$  as an increment along the line search direction  $\vec{d}$ ;
    if ( $\bar{\alpha}^{(k)}$  exist)
         $h := \min(\bar{\alpha}^{(k)}, \alpha)$ ;
    else
         $h := \epsilon$ ; //  $\epsilon$  is a small pre-defined constant.
(f)  $\vec{v}^{(k+1)} := \vec{v}^{(k)} + h \cdot \vec{d}$ ;
    while ( $\phi^*(\vec{v}^{(k+1)}) \geq \phi_{old}^*$ ) do
        REDUCTION( $h$ );
         $\vec{v}^{(k+1)} = \vec{v}^{(k)} + h \cdot \vec{d}$ ;
(g) if ( $h = \bar{\alpha}$ )
    let  $p$  be a constraint index which holds  $\bar{\alpha}$  in (e);
    add  $\vec{a}_p$  to  $\mathcal{A}$ ; // This is to add  $c_p$  into the active set.
(h)  $\phi_{old}^* = \phi^*(\vec{v}^{(k+1)})$ ;
     $k = k + 1$ ;
    goto (c);

```

□

Figure 4. Constrained graph layout algorithm.

i.e. remove a_q from \mathcal{A} , then go to step (c). If \vec{d} in step (c) is not small enough, step (e) and step (f) are used to work out a new feasible solution $\vec{v}^{(k+1)} = \vec{v}^{(k)} + h \cdot \vec{d}$. In step (e), to make sure that h will keep $\vec{v}^{(k+1)}$ feasible for (5), h is chosen to be the minimum of α and $\bar{\alpha}$ if $\bar{\alpha}$ can be found. Otherwise h is set to a pre-defined small constant ϵ if $\bar{\alpha}$ does not exist. The **while** loop in step (f) is designed to guarantee that $\phi^*(\vec{v}^{(k+1)}) < \phi^*(\vec{v}^{(k)})$. This is how the new feasible solution is produced. If $\bar{\alpha}$ has finally been chosen as an increment h in step (e) and step (f), then the c_i that holds $\bar{\alpha}$ is added to the active set, and before starting a

new iteration from steps (c) to (h), recomputes $\phi^*(\vec{v}^{(k+1)})$ and k . For further details about the *active set method*, see [17].

After variable elimination, (6) can be reduced to an unconstrained optimization problem, and any appropriate unconstrained optimization technique can be applied in step (c) to solve it. We initially use the gradient descent method and then the conjugate gradient descent method, which is more efficient when a point is close to a minimum.

The most important consideration when using the active set method is how to compute the initial feasible solution in step (b) of Figure 4. If the initial solution is close to the global optimum, then convergence of the active set algorithm will be fast. Conversely, if the initial solution is too far from the global minimum, convergence may be slow, and the algorithm may return a local minimum rather than the global minimum.

Kamada’s algorithm for unconstrained minimization of ϕ_A also requires a good initial solution. In this case, however, because there are no constraints any assignment to the variables is feasible and so it is easier to find a good “guess” for the initial solution. If there are n nodes, Kamada’s algorithm uses the assignment, θ_{init} , which assigns values to the nodes x and y coordinates so that the nodes are placed on the vertices of the regular n -polygon circumscribed by the largest circle which can fit in the display rectangle [28].

Unfortunately, in our case we cannot directly use θ_{init} as the initial solution since it may not be feasible. Also, we would like to use the suggested values for variables wherever possible. This suggests that we use as the initial solution the assignment, ψ_{init} , defined by

$$\psi_{init}(v) = \begin{cases} \psi(v) & \text{if } w(v) \neq 0, \\ \theta_{init}(v) & \text{otherwise} \end{cases}$$

where ψ is the suggested value for the node placements. However, we still cannot use ψ_{init} as the initial solution since it may not be feasible. Instead we compute the closest solution to ψ_{init} which satisfies the constraints. In other words, our initial solution, ψ'_{init} , is the solution to the following constrained optimization problem:

$$\begin{aligned} & \text{minimize} && \sum_{v \in \vec{v}} (v - \psi_{init}(v))^2 \\ & \text{subject to} && \mathbf{C}. \end{aligned} \tag{7}$$

Problem (7) is an example of a *Quadratic Programming* problem. Such problems are well-behaved since they are convex, so that the local minimum is the global minimum. There are many fast algorithms for solving quadratic programming problems. For example, interior point methods with guaranteed polynomial worst-case behavior could be used. We chose to use a specialized active set method which has been found to be fast in practice [22]. A good initial feasible solution is vital to a nice, fast graph layout and we have found the above approach to be both efficient and satisfactory.

5. Implementation for Trees

Trees are a special type of graph widely used in many different application areas. There are a variety of drawing conventions [15] for trees and many different layout algorithms and approaches have been investigated. See for example, [46, 48, 39, 27, 21]. [28] also

gives a variant of the spring model to layout trees nicely. In particular, [43] states that for the problem of producing aesthetically pleasing drawings of *binary trees*, if the coordinates assigned to the nodes are continuous variables, the problem can be solved in polynomial time. Unfortunately, none of those algorithms or approaches allow arbitrary linear constraints in tree layout. In this section we detail a constrained graph layout module for trees. Our model handles arbitrary arithmetic linear equality and inequality constraints and uses the square of the Euclidean distance as the metric over solution values.

Our model is based on viewing unconstrained tree layout as a quadratic optimization problem. Different aesthetic criteria give rise to different optimization problems. As an example of how to capture one particular aesthetic criterion for tree layout as an optimization problem, consider the layout of a *downward rooted tree*. The following constraints, C_{tree} , lay out the tree $T = \langle V, E \rangle$ with nodes $V = \{1, 2, \dots, n\}$, in an aesthetically pleasing way. By convention, v_1 is the root of the tree.

- For all x_i, x_j , if j is the right neighbor of i , add $x_j - x_i \geq g_x$ into C_{tree} , where g_x is some pre-defined minimal horizontal gap between nodes.
- For all node i with parent node j , add $y_j = y_i + g_y$ into C_{tree} where g_y is some pre-defined vertical gap between levels. And add $y_1 = root_y$ into C_{tree} , where $root_y$ is a pre-defined constant for the root y coordinate of the tree.

Let $parent(v)$ be the parent of node v and let $leftchild(w)$ be the left-most child of node w and $rightchild(w)$ be the right-most child of node w , for any node $w \in P$ where $P = \{p | p \in V \text{ and } p \text{ is an inner node}\}$. We use the objective function

$$\phi_{tree} = \sum_{j=2}^n (x_j - x_{parent(j)})^2 + \sum_{i \in P} (x_i - (x_{leftchild(i)} + x_{rightchild(i)})/2)^2$$

to capture the desire to minimize tree width, and to place a parent node close to the middle of its child(ren) as much as possible, which are often two of the most important aesthetic criteria in tree-drawing. Thus, we can layout a downward rooted tree by finding the solution to C_{tree} which minimizes ϕ_{tree} .

Our constrained tree layout module extends this idea. It is based on the following model, *Model D*, which captures the aesthetic layout of trees as a quadratic programming problem. The model is parametric in the choice of C_{tree} and ϕ_{tree} which are the constraints and objective function which capture the desired aesthetic criteria for a particular type of unconstrained tree layout. The model solves the following optimization problem:

$$\begin{aligned} \text{minimize} \quad & \phi_{tree}(\vec{v}) + \sum_{v \in \vec{v}} w_v \cdot (\psi(v) - v)^2 \\ \text{subject to} \quad & C \cup C_{tree} \end{aligned} \tag{8}$$

where $\vec{v} = (x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_n)$ and C is the input set of linear equality and inequality constraints. It is important to realize that the above definition of C_{tree} captures only one type of tree aesthetic. A weaker aesthetic might be that different levels need only be equally spaced apart.

As this is a quadratic programming problem there exist many robust and fast techniques for finding the minimum. Again we use the active set method.

Table 1. Unconstrained graph layout.

graph	Kamada's	Model A	Model B	Model C	no. of nodes
Graph 1	0.22	2.86	0.38	0.38	9
Graph 2	0.94	10.93	1.87	1.87	16
Graph 3	0.11	0.50	0.05	0.05	6
Graph 4	0.16	2.74	0.28	0.28	10
Graph 5	1.59	13.24	3.35	6.15	20
Graph 6	2.69	10.98	4.89	5.82	20
Graph 7	3.08	13.29	4.12	6.20	20
Graph 8	4.01	15.32	5.55	5.60	31
Graph 9	0.98	1.98	0.89	0.90	11

6. Empirical Evaluation

In this section we detail our empirical evaluation of the four different constrained graph layout modules. The constrained tree layout module, *Model D*, uses the aesthetic criteria given in the example for downward rooted trees. For each model we evaluate both the speed, in seconds, of the layout and the quality of the layout. All programs are implemented in *Borland C++* and run on a *DECpc LPx+ 466d2*, whose speed is 66 MHz; and the core of the program, the active set method, is approximately 1500 lines of *C++* code.

Our first experiment was to compare the quality of layout and speed of layout of *Model A*, *Model B* and *Model C* with Kamada's original algorithm for unconstrained undirected graph layout. Table 1 compares the performances of four methods. Each of the methods was tried on nine sample graphs, Graph 1 to Graph 9, of which two, Graph 1 and Graph 2, are Figure 2(b) and Figure 3(a) of [13]; Graph 3 to Graph 7 are Figure 5.9(a), Figure 5.9(b), Figure 5.11(a), Figure 5.11(c) and Figure 5.11(d) of [28], respectively; Graph 8 is taken from [41]; and Graph 9 is new.

The results in Table 1 show that *Model B* is significantly faster than *Model A* and that *Model B* is not significantly slower than Kamada's original algorithm. This demonstrates that the overhead of a general purpose constraint solving algorithm is not unreasonable even for unconstrained graph layout.

The layout of Graphs 1 to Graph 9 using *Model B* is shown in Figure 1. The quality of layout produced by *Model B* is satisfactory, and is generally similar to that produced by Kamada's algorithm (and hence *Model A*). The only important differences arise for Graph 5 and Graph 6. The layout of Graph 5 and Graph 6 using *Model B* are shown in Figure 5(e) and Figure 5(f) and should be compared with the layout given by Kamada's algorithm shown in Figure 6. Kamada's algorithm gives a nice layout, in Figure 6(a), for Graph 5 but that of *Model B* is worse, with some nodes placed too close together in Figure 5(e). For Graph 6 Kamada's algorithm gives a layout with an unnecessary edge crossing. This also happens in the layout given by *Model B*, Figure 5(f), but this time it is because node 1 is too close to node 2 and node 3 due to the weaker repulsive force, while other nodes are uniformly positioned.

Table 1 also shows that the time cost of *Model C* is significantly lower than that of *Model A*, while the disadvantage of *Model B*, that is, sometimes nodes are too close, is overcome. Taking Graph 6 for example, the layout using *Model C* is shown in Figure 7(b) and should be compared with the layout for the same graph given by Kamada's algorithm and *Model B*

Table 2. Constrained graph layout.

graph	<i>Model A</i>	<i>Model B</i>	<i>Model C</i>	no. of constraints
Graph 1	4.89	0.99	1.05	8
Graph 2	8.62	2.41	2.47	7
Graph 3	0.11	0.11	0.11	7
Graph 4	1.26	0.33	0.33	6
Graph 5	12.14	3.9	9.88	8
Graph 6	19.94	4.61	4.67	9
Graph 7	13.68	6.21	9.28	5
Graph 8	23.02	9.00	9.06	14
Graph 9	4.56	3.63	3.73	7

shown in Figure 6(b) and Figure 5(f), respectively. We can see that the edge crossing occurred in Figure 5(f) and Figure 6(b) is avoided in the layout given by *Model C*, while the effect of the weaker repulsive force, causing that node 1 to be too close to node 2 and node 3, as occurred in Figure 5(f) by *Model B*, is improved, see Figure 7(b). The node closenesses happened in Figure 5(e) for Graph 5 are successfully removed in that given by *Model C* shown in Figure 7(a).

Our second experiment is to compare *Model A*, *Model B* and *Model C* for constrained graph layout. In this experiment we have added several constraints to the example graphs from the first experiment. The constraints require some nodes to be aligned vertically and/or horizontally, to be higher (lower) than some other nodes and, to be to the left (right) of some other nodes, and so on. For examples, Graph 2, see Figure 8(b), has been given constraints asking that node 1, node 2, node 3 and node 4 are horizontally aligned, node 1, node 5, node 9 and node 13 are vertically aligned and node 14 is above node 13. And Graph 9 has been given constraints requiring that node 4, node 8 and node 9 align vertically, node 4 is lower than node 3, node 5 and node 9, and node 9 is lower than nodes 10 and 11, see Figure 8(i). The constrained layout of Graph 1 to Graph 9, with the constraints imposed, are given in Figure 8. Table 2 shows the time in seconds for each method to layout the constrained graph. Again, *Model B* is significantly faster than *Model A* and *Model C* is usually as fast as *Model B*. The constrained layouts produced by *Model B* are aesthetically pleasing.

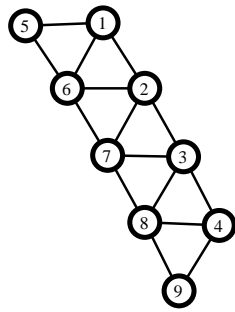
Our third experiment was to evaluate the constrained tree layout module. We took eight sample trees: Tree 1 is Figure 2 of [39], Tree 2 is Figure 1 of [48], Tree 3 and Tree 4 are Figure 2 and Figure 10 of [44] respectively, Tree 5 and Tree 6 are Figure 12 and Figure 18 from [27] respectively, and Tree 7 and Tree 8 are a four-level complete binary tree and a three-level complete 3-ary tree, respectively. We laid them out with *Model D* and also with *Model A* and *Model B*, all constrained under C_{tree} defined in Section 5. Table 3 shows the time taken to layout the trees with each method. Figure 9 is the layout of Tree 1 and Tree 2, drawn by *Model A*, *Model B* and *Model D* respectively. In addition, the *Model D*, *constrained* column in Table 3 gives the time taken to layout each tree, by *Model D*, with constraints set C which asks every non-leaf node to be vertically aligned with its left-most son, and the layout of Tree 1 and Tree 2 so constrained are given in Figure 10. Our results demonstrate that *Model D* performs better layout because of width-minimization, and is also significantly faster than the more general *Model B*.

We now take two examples to explain further how suggested values and constraints work. The first example is taken from Figure 1 of Section 1 to illustrate how suggested values work. Suppose the suggested values are the old position coordinates. As we have known that Figure 1(c) is formed by editing Figure 1(a) interactively, and then the user thinks that this is close to the layout that they feel like that is Figure 1(d), and lay it out, by *Model B*, with the suggested values mentioned above. The layout obtained is shown in Figure 11(a). However if no suggested values were given here, the layout by *Model B* would have been the same as that of Figure 11(b). The same graph layout by Kamada's original algorithm is shown in Figure 11(c), which would also destroy the user's mental map.

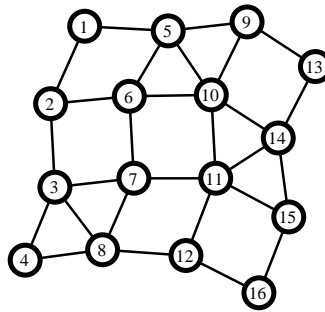
Our second example is the telecommunication network graph from Figure 2. Again, a terminal station is represented by circle, and switching centre by solid disc. The unconstrained layouts of this graph given by both Kamada's original algorithm and *Model C* are Figure 12(a) and Figure 12(b), respectively, which are unable to reflect semantics and are not adequate for this application. For a given rectangle, imposing constraints requiring that node 1 and node 2 are on the right boundary of the rectangle, node 3, node 4 and node 5 on the bottom, node 6 on the left, node 7 and node 8 on the top, and that all nodes representing switching centres must be strictly inside rectangle, the layout so constrained by *Model C* is given in Figure 12(c). Note that only terminal stations (Node 1 to Node 7) are placed on the edge of the display rectangle.

Table 3. Tree layout.

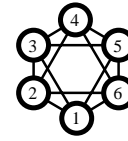
graph	<i>Model A</i>	<i>Model B</i>	<i>Model D</i>	<i>Model D</i> , constrained	no. of nodes
Tree 1	82.99	36.47	3.74	1.59	31
Tree 2	17.74	7.53	2.25	0.44	18
Tree 3	4.78	3.40	0.66	0.33	13
Tree 4	11.42	6.86	1.21	0.33	16
Tree 5	5.11	2.58	0.66	0.27	15
Tree 6	1.98	1.04	0.33	0.22	10
Tree 7	4.39	2.30	0.50	0.22	15
Tree 8	5.16	2.31	0.55	0.17	13



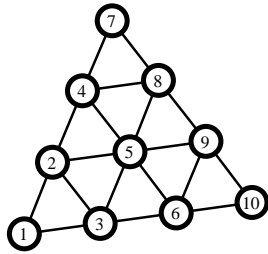
(a) Graph 1



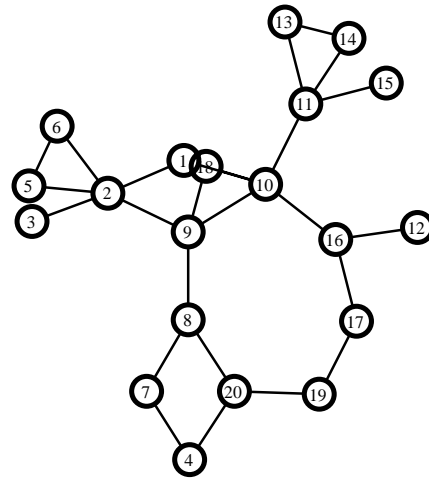
(b) Graph 2



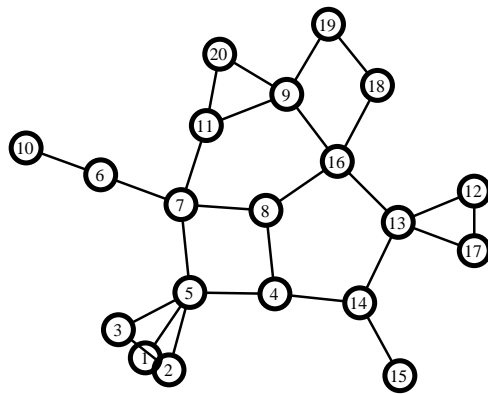
(c) Graph 3



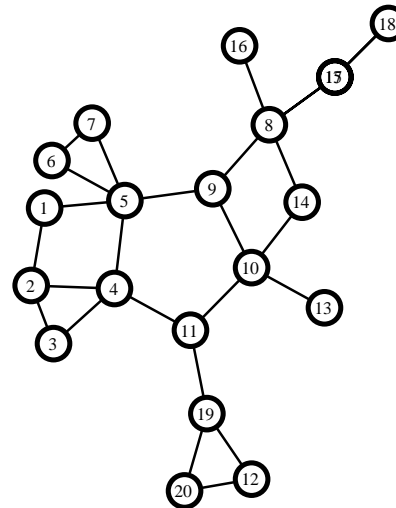
(d) Graph 4



(e) Graph 5

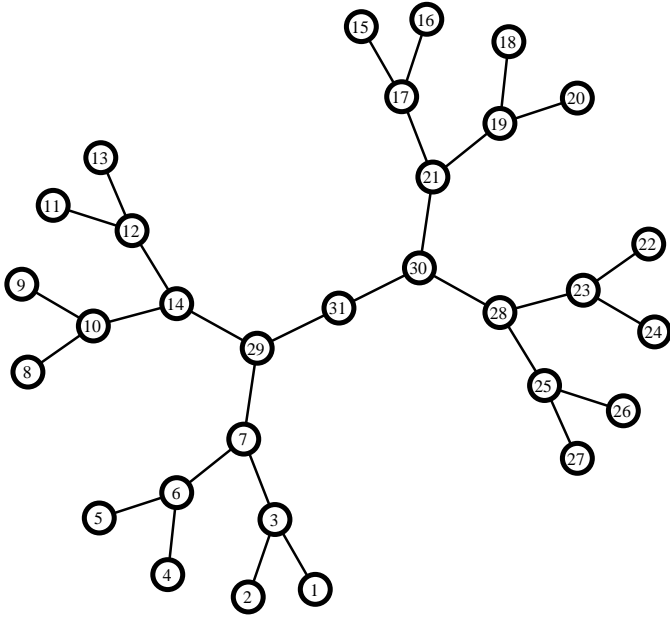


(f) Graph 6

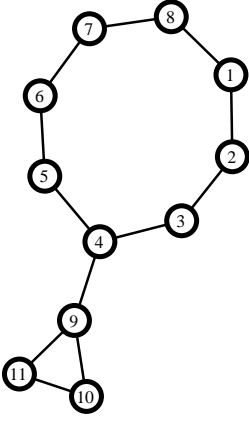


(g) Graph 7

continued



(h) Graph 8



(i) Graph 9

Figure 5. Unconstrained layout by Model B.

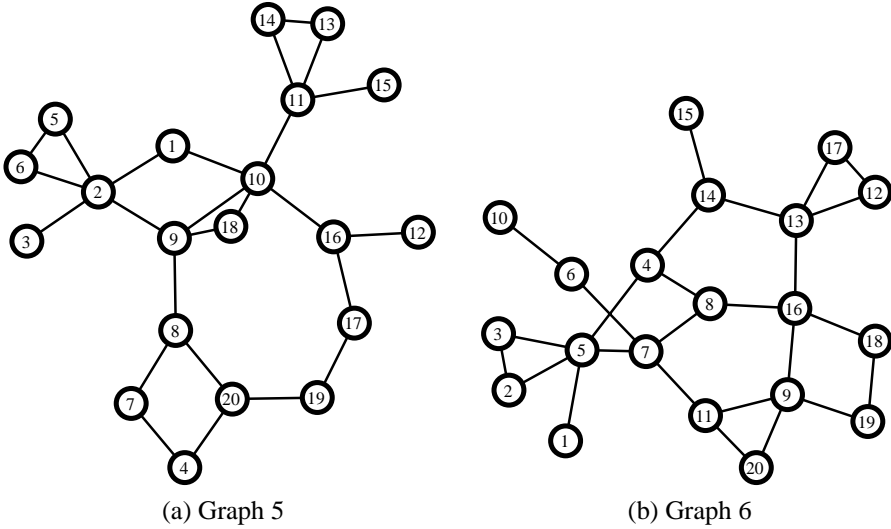


Figure 6. Layout by Kamada's algorithm.

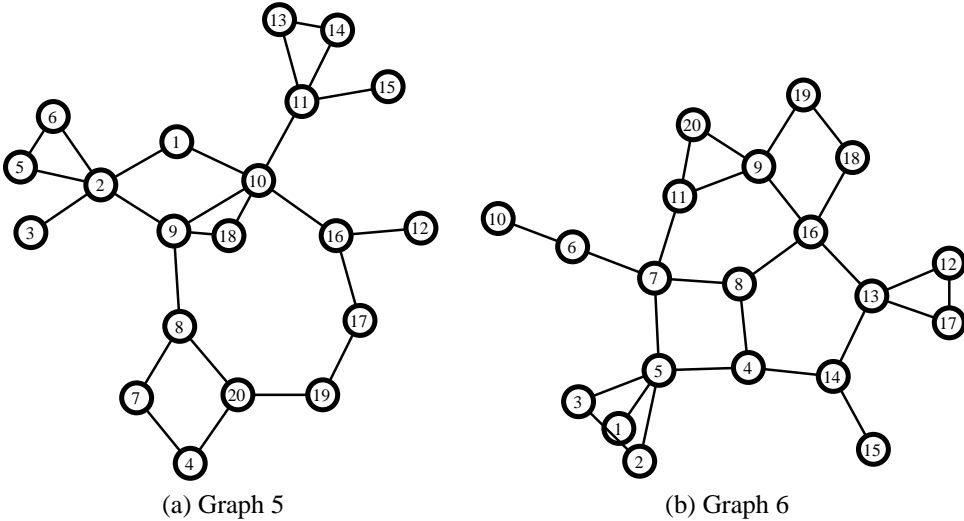
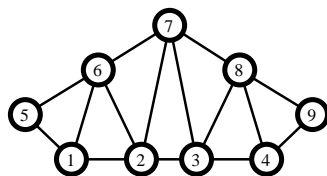
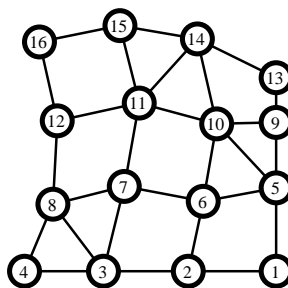


Figure 7. Unconstrained layout by Model C.



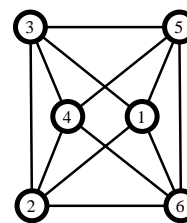
$$\begin{aligned}
 &y_1 = y_2 = y_3 = y_4 \\
 &y_6 = y_8, y_5 = y_9 \\
 &y_6 - y_7 \geq 30 \\
 &y_5 - y_6 \geq 30 \\
 &y_1 - y_5 \geq 30
 \end{aligned}$$

(a) Graph 1



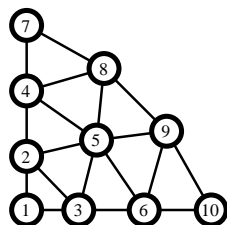
$$\begin{aligned}
 &x_1 = x_5 = x_9 = x_{13} \\
 &y_1 = y_2 = y_3 = y_4 \\
 &y_{13} - y_{14} \geq 26
 \end{aligned}$$

(b) Graph 2



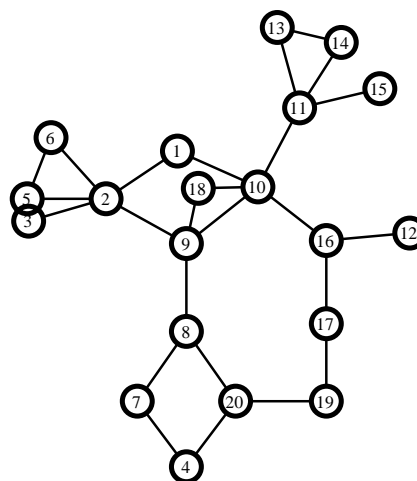
$$\begin{aligned}
 &x_1 - x_4 \geq 50 \\
 &y_3 = y_5 \\
 &y_2 = y_6 \\
 &x_5 - x_3 = 100 \\
 &x_6 - x_2 = 100 \\
 &y_6 - y_5 = 120 \\
 &y_1 = y_4
 \end{aligned}$$

(c) Graph 3



$$\begin{aligned}
 &x_1 = x_2 = x_4 = x_7 \\
 &y_1 = y_3 = y_6 = y_{10}
 \end{aligned}$$

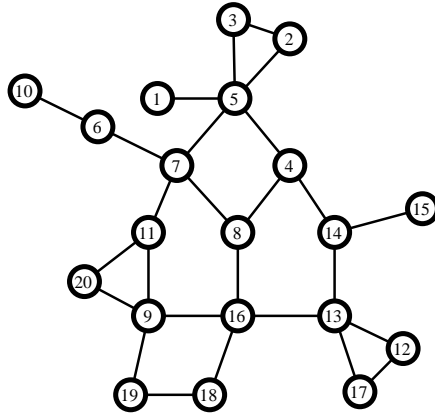
(d) Graph 4



$$\begin{aligned}
 &x_{16} = x_{17} = x_{19}, x_4 = x_8 = x_9 \\
 &y_7 = y_{19} = y_{20}, y_{18} - y_1 \geq 25 \\
 &x_4 = (x_7 + x_{20})/2
 \end{aligned}$$

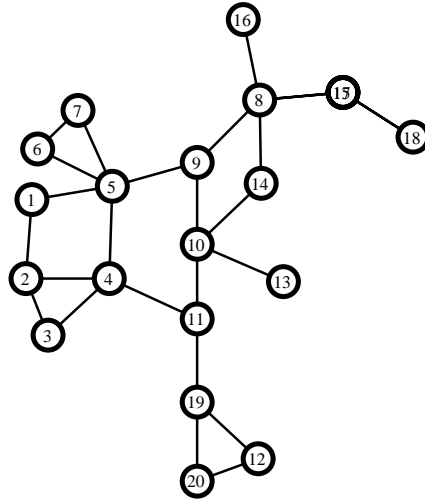
(e) Graph 5

continued



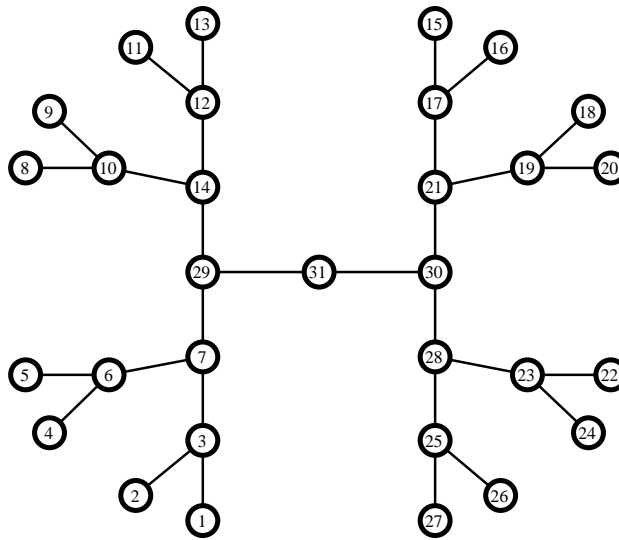
$$\begin{aligned}
 x_4 &= x_8 = x_9 \\
 x_8 &= x_{16}, x_9 = x_{11}, x_{13} = x_{14} \\
 y_9 &= y_{13} = y_{16}, y_8 = y_{11} = y_{14} \\
 y_4 &= y_7, y_1 = y_5
 \end{aligned}$$

(f) Graph 6



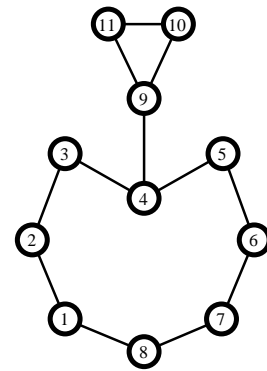
$$\begin{aligned}
 x_9 &= x_{10} = x_{11} = x_{19} = x_{20} \\
 y_{18} - y_8 &\geq 25
 \end{aligned}$$

(g) Graph 7



$$\begin{aligned}
 x_1 &= x_3 = x_7 = x_{12} = x_{13} = x_{14} = x_{29} \\
 x_{15} &= x_{17} = x_{21} = x_{25} = x_{27} = x_{28} = x_{30} \\
 y_{29} &= y_{30} = y_{31}
 \end{aligned}$$

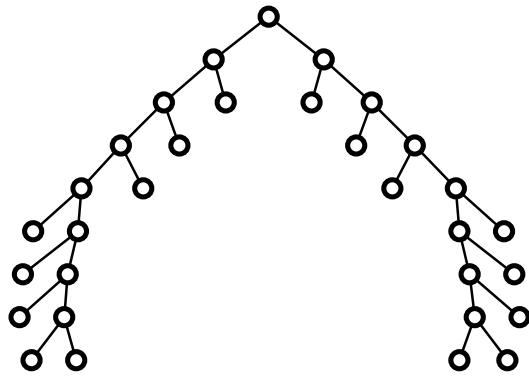
(h) Graph 8



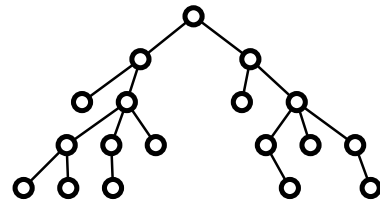
$$\begin{aligned}
 x_4 &= x_8 = x_9 \\
 y_4 - y_9 &\geq 30 \\
 y_4 - y_3 &\geq 30 \\
 y_4 - y_5 &\geq 30 \\
 y_9 - y_{11} &\geq 30 \\
 y_9 - y_{10} &\geq 30
 \end{aligned}$$

(i) Graph 9

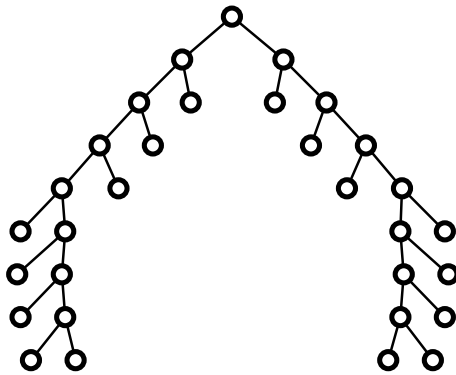
Figure 8. Constrained layout by Model B.



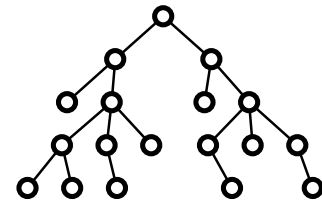
(a) Tree 1 by *Model A*



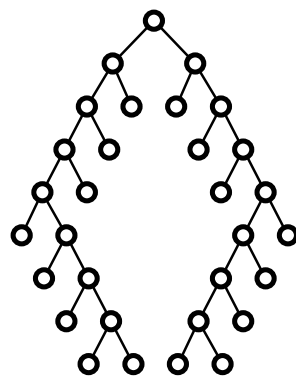
(b) Tree 2 by *Model A*



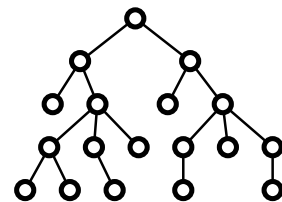
(c) Tree 1 by *Model B*



(d) Tree 2 by *Model B*



(e) Tree 1 by *Model D*



(f) Tree 2 by *Model D*

Figure 9. Unconstrained tree layout by *Model A, B and D.*

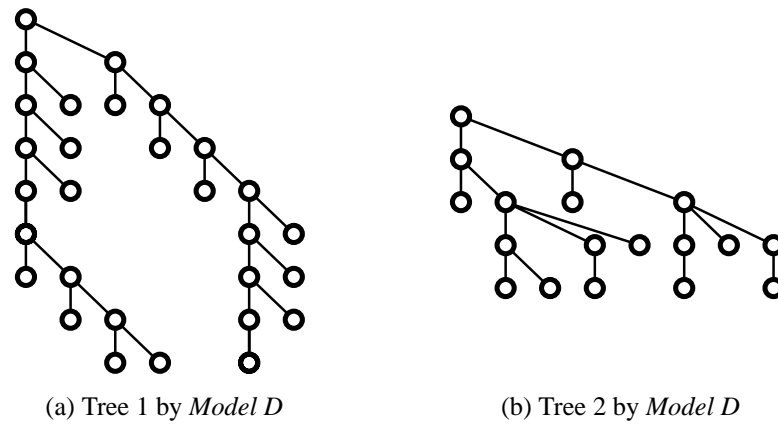


Figure 10. Constrained tree layout by *Model D*.

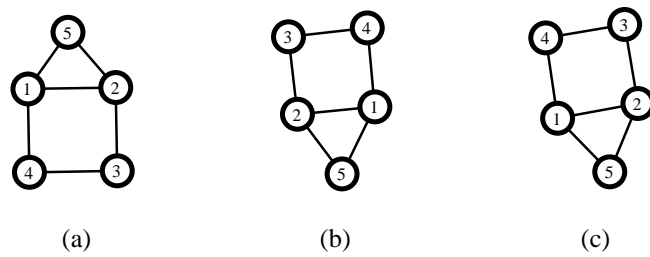


Figure 11. Example of interaction with suggested values presenting.

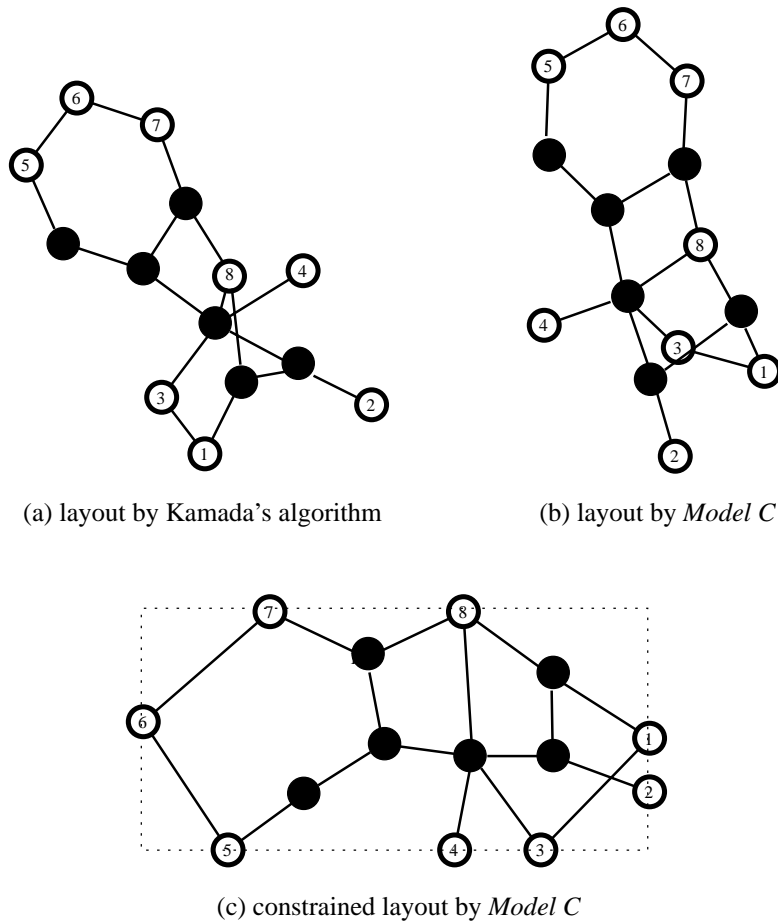


Figure 12. Example of constrained telecommunication network graph layout.

7. Conclusion and Future Work

We have introduced a generic model – constrained graph layout – for interactive graph layout and described four implementations of the model, three for undirected graph layout and one for tree layout. The key feature of our new model is that it allows the user to constrain the position of nodes in the graph and to provide suggested values for the node locations. The greater flexibility of our model does not come at a high computational cost. Empirical evaluation of the implementations shows that the overall time cost of our models, apart from *Model A*, is not significantly lower than that of Kamada's original algorithm. In particular our second implementation, *Model B*, provides good layout of undirected graphs in the context of arbitrary linear constraints at a reasonable cost, while *Model C* produces

layout with the benefits of both *Model A* and *Model B*. Our *Model D* provides quick and reasonable layout of trees in the context of arbitrary linear constraints.

The main motivation for our work is from work in advanced visual interfaces. First, constrained graph layout can be used in animation in which the new diagram is defined in terms of objects and constraints in the old diagram, and in which remaining objects in the diagram should not be moved unless necessary. The second use is for user interfaces for pen-based computing. One important approach in such user interfaces is to parse the pen-drawn diagram to infer constraints between the components [8]. Constrained graph layout can be used to re-layout or “pretty print” the recognized diagram while preserving its semantics. Constrained graph layout also has numerous other applications in user interfaces – basically whenever the diagram has more semantic structure than a simple graph, then the constrained graph layout model is appropriate.

In the future, we intend to develop a new constrained graph layout module which is for directed graph layout. Typically, directed graph layout has four stages – layering, ordering, positioning and edge drawing [16, 20, 42]. We believe that we can modify the first three stages so as to handle appropriate user specified constraints. Also, the idea for mental map preservation in our models suggests a general foundation for solving the problem of constrained layout adjustment and mental map preservation.

Acknowledgments

We think Yi Xiao for permission to use her quadratic solver and Sitt Sen Chok for his visual editor code.

Notes

- Essentially, the minimum of a function $f(x_1, x_2, \dots, x_n)$ subject to equality constraints $e_j(x_1, x_2, \dots, x_n) = 0$ for $j = 1, 2, \dots, s$, is to be found among the turning points of the *Lagrangian form*

$$\Phi(\mathbf{x}, \lambda) = f(\mathbf{x}) + \sum_{j=1}^s \lambda_j \cdot e_j(\mathbf{x})$$
where $\lambda = (\lambda_1, \dots, \lambda_s)$ are known as Lagrange multipliers [4].

References

- G. D. Battisa, P. Eades, R. Tamassia, and I. G. Tollis. (1994). Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry: Theory and Applications*, 4:235–282.
- K. Böhringer and F. N. Paulisch. (1990). Using constraints to achieve stability in automatic graph layout algorithms. In *Proc. of Conference on Human Factors in Computing Systems, CHI'90*, pages 43–51.
- A. Borning. (1981). The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):252–387.
- M. J. Box, D. Davies, and W. H. Swann. (1969). *Non-linear Optimization Techniques*. Oliver & Boyd.
- F. J. Brandenburg. (1994). Designing graph drawings by layout graph grammars. In *Proceedings of DIMACS International Workshop, GD'94, LNCS 894*, pages 416–427, Princeton, New Jersey, USA. Springer-Verlag.
- F. J. Brandenburg, M. Himsolt, and C. Rohrer. (1995). An experimental comparison of force-directed and randomized graph drawing algorithms. In *Symposium on Graph Drawing, GD'95, LNCS 1027*, pages 76–87, Passau, Germany. Springer-Verlag.
- K. W. Cattermole. (1979). Graph theory and communications networks. In R. J. Wilson and L. W. Beineke, editors, *Applications of Graph Theory*, pages 17–57. Academic Press.

8. S. S. Chok and K. Marriott. (1995). Automatic construction of user interfaces from constraint multiset grammars. In *IEEE Symposium on Visual Languages*, pages 242–249.
9. P. Kikusts and P. Ručevskis. (1995). Layout algorithm of graph-like diagrams for grade windows graphic editors. In *Symposium on Graph Drawing, GD'95, LNCS 1027*, pages 361–364, Passau, Germany. Springer-Verlag.
10. I. F. Cruz and A. Garg. (1994). Drawing graphs by example efficiently: trees and planar acyclic digraphs. In *Proceedings of DIMACS International Workshop, GD'94, Princeton, New Jersey, USA, October 1994, LNCS 894*, pages 404–415. Springer-Verlag.
11. R. Davidson and D. Harel. (1991). Drawing graphs nicely using simulated annealing. Technical report, Department of Applied Mathematics and Computer Science.
12. E. Dengler, M. Friedell, and J. Marks. (1993). Constraint-driven diagram layout. In *Proceedings of the 1993 IEEE Symposium on Visual Languages*, pages 330–335.
13. P. Eades. (1984). A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160.
14. P. Eades, W. Lai, K. Misue, and K. Sugiyama. (1991). Preserving the mental map of a diagram. In *Proceedings of Compugraphics '91*, pages 24–33.
15. P. Eades, T. Lin, and X. Lin. (1990). Two tree drawing conventions. Technical Report 174, Key Centre for Software Technology, Department of Computer Science, The University of Queensland.
16. P. Eades and K. Sugiyama. (1990). How to draw a directed graph. *Journal of Information Processing*, 13(4):424–437.
17. R. Fletcher. (1987). *Practical Methods of Optimization*. John Wiley & Sons.
18. A. Frick, A. Ludwig, and H. Mehldau. (1994). A fast adaptive layout algorithm for undirected graphs. In *Proceedings of DIMACS International Workshop, GD'94, LNCS 894*, pages 388–403, Princeton, New Jersey, USA. Springer-Verlag.
19. T. M. J. Fruchterman and E. M. Reingold. (1991). Graph drawing by force-directed placement. *Software-Practice and Experience*, 21(11):1129–1164.
20. E. R. Gansner, E. Koutsofios, S. C. North, and K. Vo. (1993). A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230.
21. A. Garg, M. T. Goodrich, and R. Tamassia. (1994). Area-efficient upward tree drawing. In *Proceedings of the 9th Annual Symposium on Computational Geometry, ACM*, pages 359–368.
22. D. Goldfarb and A. Idnani. (1983). A numerically stable dual method for solving strictly convex quadratic programs. *Math. Prog.*, 27:1–33.
23. W. He and K. Marriott. (1997). Constrained graph layout. In *Proceedings of International Symposium on Graph Drawing, GD'96, LNCS 1190*, pages 217–232, Berkeley, California, USA. New York:Springer.
24. R. Helm and K. Marriott. (1986). Declarative graphics. In *Proc. of the 3rd International Conference on Logic Programming, LNCS 225*, pages 513–527, London, England. Springer-Verlag.
25. R. Helm and K. Marriott. (1991). A declarative specification and semantics for visual languages. *Journal of Visual Languages and Computing*, 2:311–331.
26. R. Helm, K. Marriott, T. Huynh, and J. Vlissides. (1995). An object-oriented architecture for constraint-based graphical editing. In *Object-Oriented Programming for Graphics*, pages 217–238. Springer-Verlag.
27. J. Q. Walker II. (1990). A node-position algorithm for general tree. *Software-Practice and Experience*, 20(7):685–705.
28. T. Kamada. (1989). *Visualizing Abstract Objects and Relations: a Constraints-based Approach*, volume 5 of *Computer Science*. Singapore, New Jersey:World Scientific.
29. T. Kamada and S. Kawai. (1989). An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15.
30. T. Kamps, J. Kleinz, and J. Read. (1995). Constraint-based spring-model algorithm for graph layout. In *Symposium on Graph Drawing, GD'95, LNCS 1027*, pages 349–360, Passau, Germany. Springer-Verlag.
31. T. Lin and P. Eades. (1994). Integration of declarative and algorithmic approaches for layout creation. Technical Report TR-HJ-94-10, CSIRO Division of Information Technology, Centre for Spatial Information Systems.
32. P. Lüders, R. Ernst, and S. Stille. (1995). An approach to automatic display layout using combinatorial optimization. *Software-Practice and Experience*, 25(11):1183–1202.
33. K. A. Lyons. (1992). Cluster busting in anchored graph drawing. In *Proceedings of CASCON '92*, pages 7–17, Toronto.
34. K. Misue, P. Eades, W. Lai, and K. Sugiyama. (1995). Layout adjustment and the mental map. *Journal of Visual Languages and Computing*, 6:183–210.

35. B. A. Myers, D. A. Giuse, R. B. Dannenberg, B. V. Zanden, D. S. Kosbie, E. Pervin, A. Mickish, and P. Marchal. (1990). Garnet: comprehensive support for graphical highly interactive user interfaces. *Comput.*, pages 71–85.
36. S. C. North. (1995). Incremental layout in DynaDAG. In *Symposium on Graph Drawing, GD'95, LNCS 1027*, pages 409–418, Passau, Germany. Springer-Verlag.
37. A. Papakostas, J. M. Six, and I. G. Tollis. (1997). Experimental and theoretical results in interactive orthogonal graph drawing. In *Proceedings of International Symposium on Graph Drawing, GD'96, LNCS 1190*, pages 371–386, Berkeley, California, USA. New York:Springer.
38. F. N. Paulisch. (1993). *The Design of an Extendible Graph Editor*. Springer-Verlag. LNCS 704.
39. E. M. Reingold and J. S. Tilford. (1981). Tidier drawing of trees. *IEEE Transaction on Software Engineering*, SE-7(2):223–228.
40. M. D. Storey and H. A. Müller. (1995). Graph layout adjustment strategies. In *Symposium on Graph Drawing, GD'95, LNCS 1027*, pages 487–499, Passau, Germany. Springer-Verlag.
41. K. Sugiyama and K. Misue. (1995). Graph drawing by magnetic-spring model. *J. Visual Lang. Comput.*, 6(3). Special issue on Graph Visualization edited by I.F. Cruz and P. Eades.
42. K. Sugiyama, S. Tagawa, and M. Toda. (1981). Methods for visual understanding of hierarchical system structures. *IEEE Transaction on Systems, Man, and Cybernetics*, SMC-11(2):109–125.
43. K. J. Supowit and E. M. Reingold. (1983). The complexity of drawing trees nicely. *Acta Informatica*, 18:377–392.
44. K. Tsuchida, Y. Adachi, Y. Oi, Y. Miyadera, and T. Yaku. (1995). Constraints and algorithms for drawing tree-structured diagrams. In *Proceedings of the International Workshop on Constraints for Graphics and Visualization, CGV '95*, pages 87–101, Cassis, France.
45. D. Tunkelang. (1994). A practical approach to drawing undirected graphs. Carnegie Mellon University.
46. J. G. Vaucher. (1980). Pretty-printing of trees. *Software-Practice and Experience*, 10:553–561.
47. L. Weitzman and K. Wittenburg. (1993). Relation grammars for interactive design. In *Proceedings of IEEE Visual Languages*, pages 4–11.
48. C. Wetherell and A. Shannon. (1979). Tidy drawing of trees. *IEEE Transaction on Software Engineering*, SE-5(5):514–520.