Constrained Graph Layout

Weiqing He and Kim Marriott

Computer Science Department,
Monash University,
Clayton 3168, Australia

{whe,marriott}@cs.monash.edu.au

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 layed out. We describe three implementations of layout modules which are based on the constrained graph layout model. The first two implementations are for undirected graph layout and the third 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.

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 preserve the *mental map* of the user [11, 32, 28], that is the new layout should not move an existing node unless the current position leads to poor layout. The second reason is that almost all existing graph layout algorithms are quite restrictive in how graphs can be laid out since they encapsulate fixed layout aesthetics. The application program cannot place constraints on the layout which take into account the underlying semantics of the object represented by the graph.

To overcome these problems, we introduce a general model of *constrained graph layout* which is better suited for interactive applications involving graph layout, see Fig. 1. 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.

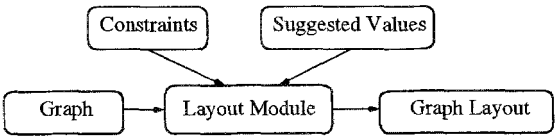

Fig. 1. The Constrained Graph Layout Model

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 three different constrained graph layout modules. All three modules allow arbitrary linear arithmetic constraints. The first two modules are for undirected connected graphs, while the third module is for tree layout. The first module is based on Kamada's aesthetic cost function [22, 23]. The second module is a modification of this cost function which removes the non-polynomial terms. Unfortunately, Kamada's algorithm for graph layout is not suitable for constrained graph layout since it cannot handle interaction between variables. Instead we use active set methods 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 three modules. Our evaluation shows that our second layout module for undirected connected graphs is not significantly slower 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 model based on Kamada's original cost aesthetic. Third, we show that the tree layout module is significantly faster than the other two modules. Thus, if you are only laying out trees you should use this specialized module.

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 [24]. 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 nodes. Furthermore, they do not allow suggested values for nodes. Other closely related work is described in Luders *et al* [27] 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* [9] 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.

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 [4], Helm and Marriott [19, 18], Weitzman and Wittenburg [38] and Cruz *et al* [7]. There is also a considerable body of literature on constraint based diagram manipulation. See for instance, Garnet [29], QOCA [20] and ThingLab [2]. More general related work is described in the surveys [12, 1]. In particular, force-directed graph drawing algorithms for unconstrained graph layout can be found in [8, 23, 36, 15, 34, 10, 14], and [5] contains an experimental comparison between [36, 14, 8, 23, 15]. Algorithms for incremental unconstrained graph layout are given in [25, 30], while [26] discusses the approach of integration of declarative and algorithmatic graph layout.

2 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 [11, 32] 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 Fig. 1. The first is a *graph*, $G = \langle V, E \rangle$, where $V = \{1, \ldots, n\}$ 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:

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

where $\mathbf{v} = (x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n)$ and $\phi(\mathbf{v})$ captures the graph layout aesthetic as an objective function over the nodes' x coordinates and y coordinates, $C(\mathbf{v})$ 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 criteria 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.

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 three different graph layout modules which we have implemented.

3 Implementation for Undirected Graphs

In this section we detail two constrained graph layout models for general undirected connected graphs. Both 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 [23]. 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 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.

We can rewrite this cost function to the following:

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

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.

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 squared differences between the desired distance between nodes and the actual distance. 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 has no definition 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* [13], the increment in line search can be determined symbolically if the aesthetic cost function has polynomial form while only numeric methods can be used if the aesthetic cost function includes $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ terms.

The most important question is how we can efficiently solve the resulting optimization problems when using *Model A* or *Model B* for constrained graph layout. In both cases we must solve a constrained optimization problem of the following form:

$$\begin{array}{ll} \textit{minimize} & \phi^*(\mathbf{v}) \\ \textit{subject to} & \mathbf{C} \end{array} \tag{4}$$

where: $\mathbf{v} = (x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n)$;
$\mathbf{C}$ is a set of linear equality and inequality constraints, and
$\phi^*(\mathbf{v})$ is $\phi_A(\mathbf{v}) + \sum_{v \in \mathbf{v}} w(v) \cdot (\psi(v) - v)^2$ in the case of *Model A*, and
$\phi^*(\mathbf{v})$ is $\phi_B(\mathbf{v}) + \sum_{v \in \mathbf{v}} w(v) \cdot (\psi(v) - v)^2$ in the case of *Model B*.

Kamada [22] gives a simple and efficient algorithm to minimize ϕ_A in the case that there are no constraints. Kamada 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 (4) because the constraints introduce interaction between variables. This means 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 (4).

Our implementation for *Model A* and *Model B* is instead based on the *Active Set Method* [13]. This is an iterative technique developed by the operations research community 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, ..., O_n, which only have equality constraints. This set of equality constraints, $\mathcal{A}$, is

(a) Compute l_{ij}, k_{ij} for $1 \leq i \neq j \leq n$;
(b) Compute initial feasible solution $\mathbf{v}^{(1)}$ and active set $\mathcal{A} := \mathcal{A}(\mathbf{v}^{(1)})$;
$\phi^*_{old} = \phi^*(\mathbf{v}^{(1)})$;
(c) Compute $\mathbf{d}$ by solving (5);
if $(\mathbf{d} \geq \varepsilon)$ **goto** (e);
(d) Let $\lambda_q^{(k)}$ solve *min* $\lambda_i^{(k)}$, for all $\mathbf{a}_i \in \mathcal{A}$ and c_i is an inequality constraint;
if $(\lambda_q^{(k)} \geq 0)$
terminate with $\mathbf{v}^{(k)}$ as the solution;
else{
remove $\mathbf{a}_q$ from $\mathcal{A}$;
goto (c); }
(e) Compute $\overline{\alpha}^{(k)} := \min \frac{b_i - \mathbf{v}^{(k)} \cdot \mathbf{a}_i}{\mathbf{d} \cdot \mathbf{a}_i}$ for $\mathbf{a}_i$ in $\overline{\mathcal{A}}$ and $\mathbf{d} \cdot \mathbf{a}_i < 0$
Choose α as an increment along the line search direction $\mathbf{d}$;
if ($\overline{\alpha}^{(k)}$ *exist*)
$h := \min(\overline{\alpha}^{(k)}, \alpha)$;
else
$h := \alpha$;
(f) $\mathbf{v}^{(k+1)} := \mathbf{v}^{(k)} + h \cdot \mathbf{d}$;
while $(\phi^*(\mathbf{v}^{(k+1)}) \geq \phi^*_{old})$ {
REDUCTION(h);
$\mathbf{v}^{(k+1)} = \mathbf{v}^{(k)} + h \cdot \mathbf{d}$; }
(g) **if**($h = \overline{\alpha}$) {
let p be a constraint index which holds $\overline{\alpha}$ in (e);
add $\mathbf{a}_p$ to $\mathcal{A}$;}
(h) $\phi^*_{old} = \phi^*(\mathbf{v}^{(k+1)})$;
$k = k + 1$;
goto (c);

Fig. 2. Constrained Graph Layout Algorithm

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 $\mathbf{d}$ reducing the objective function. However, it may be that the new solution while feasible with respect to the active set of O_i, holds $\mathbf{v} \cdot \mathbf{a} = b$ for an inequality constraint $\mathbf{v} \cdot \mathbf{a} \geq b$ of the original problem which is not in the active set. In this case the corresponding equality $\mathbf{v} \cdot \mathbf{a} = b$ is added to the active set, giving rise to a new optimization problem O_i. 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 Fig. 2. Assume that $\mathbf{C} = \{c_1, c_2, \ldots, c_m\}$

and that $\mathbf{a} = (\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_m)$ and $\mathbf{b} = (b_1, b_2, \ldots, b_m)$ where $\mathbf{a}_i$ and b_i are coefficient column vector and right-hand side constant of c_i. At each step $\mathcal{A} = \{\mathbf{a}_i : c_i \text{ is active}\}$ is a column vector set representing the active set.

The algorithm in Fig. 2 proceeds as follows. In step(b), an initial feasible solution is found, and the initial active set $\mathcal{A}$ is set to $\mathcal{A}(\mathbf{v}^{(1)})$, which is the set of $\mathbf{a}_i$ such that $\mathbf{v}^{(1)} \cdot \mathbf{a}_i = b_i$ holds. Step(c) to step(h) are then performed iteratively. In each iteration $\mathbf{v}^{(k)}$ is a feasible point and step(c) attempts to solve the optimization problem

$$\begin{aligned} &\textit{minimize} && \phi^*(\mathbf{v}^{(k)} + \mathbf{d}) \\ &\textit{subject to} && \mathbf{d} \cdot \mathbf{a}_i = 0, \mathbf{a}_i \in \mathcal{A}. \end{aligned} \tag{5}$$

The solution to this problem is the search direction $\mathbf{d}$. If $\mathbf{d}$ is small enough, that means $\mathbf{v}^{(k)}$ is an acceptable solution, and the Lagrange multipliers $\boldsymbol{\lambda}^{(k)}$ of the problem are examined to determine if a local minimum has been reached.[1] Denote $\lambda_q^{(k)} = \min \lambda_i^{(k)}$ for all $\mathbf{a}_i \in \mathcal{A}$ and c_i is an inequality constraint. The algorithm terminates with $\mathbf{v}^{(k)}$ as the solution if $\lambda_q^{(k)}$ is non-negative because if all λ_i are non-negative there exist no better solution near $\mathbf{v}^{(k)}$, then $\mathbf{v}^{(k)}$ is a local minimum. Otherwise c_q should be removed from the active set, i.e. remove a_q from $\mathcal{A}$, then go to step(c). If $\mathbf{d}$ in step(c) is not small enough, step(e) and step(f) are used to work out a new feasible solution $\mathbf{v}^{(k+1)} = \mathbf{v}^{(k)} + h \cdot \mathbf{d}$. In step(e), to make sure that h will keep $\mathbf{v}^{(k+1)}$ feasible for (4), h is chosen to be the minimum of α and $\overline{\alpha}$ if $\overline{\alpha}$ can be found. Otherwise h is set to α if $\overline{\alpha}$ does not exist. The **while** loop in step(f) is designed to guarantee that $\phi^*(\mathbf{v}^{(k+1)}) < \phi^*(\mathbf{v}^{(k)})$. This is how the new feasible solution is produced. If $\overline{\alpha}$ has finally been chosen as an increment h in step(e) and step (f), then the c_i that holds $\overline{\alpha}$ is added to the active set, and before starting a new iteration from step(c), step(h) recomputes $\phi^*(\mathbf{v}^{(k+1)})$ and k. For further details about the *active set method*, please see [13].

After variable elimination, (5) can be reduced to an unconstrained optimization problem, and then any appropriate unconstrained optimization technique can be applied in step(c) to solve it. We first use 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 Fig. 2. 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.

---

[1] Essentially, the minimum of a function $f(x_1, x_2, \ldots, x_n)$ subject to equality constraints

$$e_j(x_1, x_2, \ldots, x_n) = 0 \text{ for } j = 1, 2, \ldots, s,$$

is to be found among the turning points of the *Lagrangian form*

$$\Phi(\mathbf{x}, \lambda) = f(\mathbf{x}) + \textstyle\sum_{j=1}^{s} \lambda_j \cdot e_j(\mathbf{x})$$

where $\lambda = (\lambda_1, \ldots, \lambda_s)$ are known as Lagrange multipliers [3].

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 [23].

Unfortunately, in our case we cannot directly use θ_{init} as the initial solution as 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 cannot use ψ_{init} as the initial solution because 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:

$$\begin{aligned} &\textit{minimize} && \sum_{v \in \mathbf{v}} (v - \psi_{init}(v))^2 \\ &\textit{subject to} && \mathbf{C}. \end{aligned} \tag{6}$$

Problem (6) 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 [17]. 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.

4 Implementation for Trees

Trees are a special type of graph which are widely used in many different application areas. There are a variety of drawing conventions [31] for trees and many different layout algorithms and approaches have been investigated. See for example, [37, 39, 33, 21, 16]. In particular, [22] gives a variant of the spring model to layout trees nicely. Unfortunately, all of these algorithms and approaches are for unconstrained 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 criteria for tree layout as an optimization problem, consider the layout of a *downward rooted tree*. The following constraints, $\mathbf{C}_{tree}$, lay out the tree $T =< V, E >$, where $V = \{1, 2, \ldots, n\}$ is the set of nodes, 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 $\mathbf{C}_{tree}$, where g_x is some pre-defined minimal horizontal gap between nodes;
- for any non-leaf node i, if l is i's left-most son and r is i's right-most son, add $x_i = (x_l + x_r)/2$ into $\mathbf{C}_{tree}$;

All vertices' y-coordinates y_i are placed along horizontal lines according to their level. Let $parent(v)$ be the parent of node v. We use the objective function $\phi_{tree} = \sum_{j=2}^{n}(x_j - x_{parent(j)})^2$ to capture the desire to minimize tree width, which is often one of the most important aesthetic criteria in tree-drawing. Thus, we can layout a downward rooted tree by finding the solution to $\mathbf{C}_{tree}$ which minimizes ϕ_{tree}.

Our constrained tree layout module extends this idea. It is based on the following model, *Model C*, which captures the aesthetic layout of trees as a quadratic programming problem. The model is parametric in the choice of $\mathbf{C}_{tree}$ 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:

$$\begin{aligned} &\textit{minimize} && \phi_{tree}(\mathbf{v}) + \sum_{v \in \mathbf{v}} w_v \cdot (\psi(v) - v)^2 \\ &\textit{subject to} && \mathbf{C} \cup \mathbf{C}_{tree} \end{aligned} \tag{7}$$

where: $\mathbf{v} = (x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n)$;
$\mathbf{C}$ is the input set of linear equality and inequality constraints.

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

5 Emprical Evaluation

In this section we detail our empirical evaluation of the three different constrained graph layout modules we have described in the last two sections. The constrained tree layout module, *Model C*, 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*.

Our first experiment was to compare the quality of layout and speed of layout of *Model A*, *Model B* and Kamada's original algorithm for unconstrained undirected graph layout. Each of the three methods was tried on five sample graphs, of which two, Graph 1 and Graph 2, are Fig. 5.9(b) and Fig. 5.11(c) in [22] respectively; one, Graph 3, is taken from [10]; Graph 4, is given by us;

and the last one is K_{10}. Table 1 shows the time for each method to layout the graph. The layouts using *Model B* of Graph 2 and Graph 4 are shown in Fig. 3 and Fig. 4, respectively. The results 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 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 difference is for Graph 2. The layout using *Model B* is shown in Fig. 3 and should be compared with the layout given by Kamada's algorithm (shown in [22]). Kamada's algorithm gives a layout with an edge crossing. This is avoided by *Model B* but the layout is still not very good because node 1 is too close to node 2 and node 3 due to the weaker repulsive force, while other nodes are uniformly positioned.

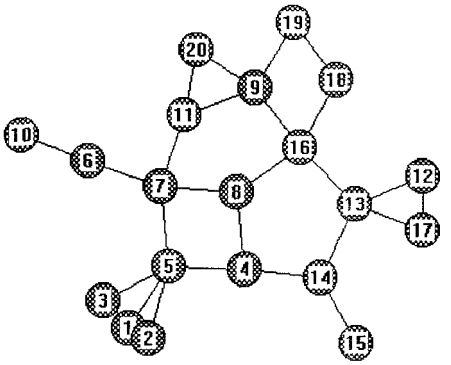

Fig. 3. Graph 2,unconstrained

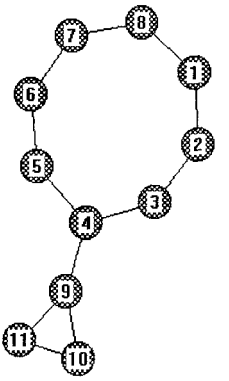

Fig. 4. Graph 4,unconstrained

graph	Kamada's	*Model A*	*Model B*	no. of nodes
Graph 1	0.17	2.08	0.39	10
Graph 2	2.75	33.12	4.89	20
Graph 3	0.77	17.03	1.48	16
Graph 4	0.55	1.59	0.93	11
Graph 5	0.23	4.33	0.16	10

Table 1. Unconstrained Graph Layout

Our second experiment is to compare *Model A* and *Model B* 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, or to be higher than some other nodes. Table 2 shows the time in seconds for each method to layout the constrained graph. Again, *Model B* is significantly faster than *Model A*. The constrained layouts produced by *Model B* are aesthetically pleasing. As examples of the quality, two of the layed out graphs are shown in Fig. 5 and Fig. 6. Graph 4 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. Note that the layout time for Graph 5 by *Model B* in Table 2 and in Table 1 is quite small. This is because for K_{10}, the initial feasible solution obtained by solving (6) is very close to a minimum. This indicates the importance of an good initial configuration, and *Model B* will be even faster than *Model A* if it is given a better initial feasible solution.

Our third experiments was to evaluate the constrained tree layout module. We took five sample trees: Tree 1 is Fig.2 of [33], Tree 2 and Tree 3 are Fig.10 and Fig.2 of [35] respectively, Tree 4 is Fig.18 of [21] and Tree 5 is a four-level complete binary tree. We lay them out with *Model C* and also with *Model A* and *Model B*, all constrained under $\mathbf{C}_{tree}$. Table 3 shows the time taken to layout the tree with each method. In addition, the *constrained* column in Table 3 gives the time taken to layout each tree, by *Model C*, with constraints which make one of the subtrees of the root an *upward rooted tree* and node gap narrower. Fig. 7 and Fig. 8 are the layout of Tree 1, drawn by *Model B* and *Model C* respectively. Our results demonstrate that *Model C* performs better layout because of width-minimization, and is also significantly faster than the more general *Model B*.

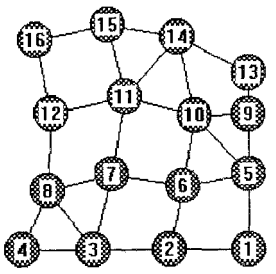

Fig. 5. Graph 3,constrained

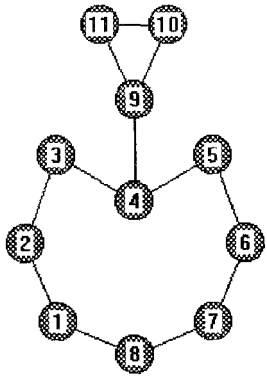

Fig. 6. Graph 4,constrained

Finally, we take Graph 4 as an example to illustrate how suggested values work. As explained above, Fig. 4 is the layout of Graph 4 using *Model B* without

graph	*Model A*	*Model B*	no. of constraints
Graph 1	5.78	1.42	4
Graph 2	38.61	14.56	9
Graph 3	18.51	2.86	7
Graph 4	9.67	4.61	7
Graph 5	5.93	0.94	2

Table 2. Constrained Graph Layout

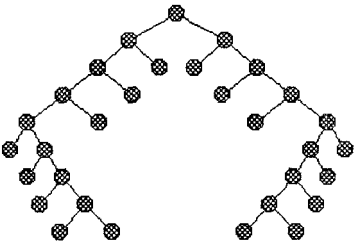

Fig. 7. Tree 1, *Model B* with C_{tree}

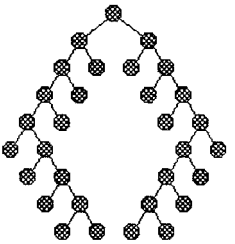

Fig. 8. Tree 1, *Model C* with C_{tree}

graph	*Model A*	*Model B*	*Model C*	constrained	vertices
Tree 1	64.60	37.35	3.41	3.62	31
Tree 2	4.50	2.91	0.82	1.05	16
Tree 3	5.22	1.65	0.55	0.82	13
Tree 4	4.23	1.37	0.55	0.71	10
Tree 5	1.59	1.53	0.61	0.82	15

Table 3. Tree Layout

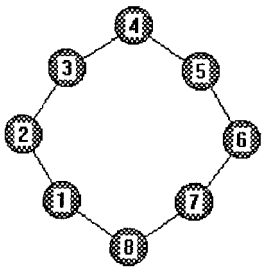

Fig. 9. phase 1

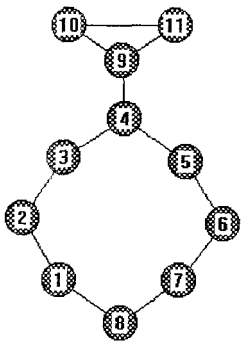

Fig. 10. phase 2

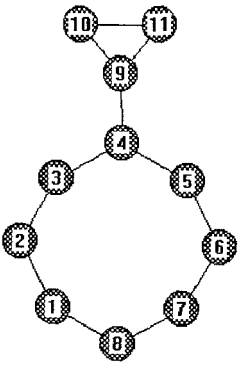

Fig. 11. phase 3

any constraints and with no suggested values. Now consider a sample interaction in which the suggested values are the old position coordinates. A user draws a graph and gets the unconstrained layout shown in Fig. 9. The user then edits the graph by adding three nodes and four arcs to a graph as displayed in Fig. 10. The user then thinks that this is close to the layout that he feels like and lays it out with the suggested values above mentioned. The layout obtained is shown in Fig. 11. Note that if no *suggested values* were given here, the layout would have been the same as that of Fig. 4.

6 Conclusion

We have introduced a generic model – constrained graph layout – for interactive graph layout and described three implementations of the model, two 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. Empirical evaluation of the implementations shows that the greater flexibility of our model does not come at a high computational cost. 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*, provides quick and reasonable layout of trees in the context of arbitrary linear constraints.

In practice, *Model A* and *Model B* can be integrated into a two-phase procedure to obtain the benefits of both models. 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, that is: shift nodes that are coincident slightly along different directions to make every node in a different point, then call *Model A* to finalize the layout. Phase two would not take too long because some local minimum should be closer, and the layout that phase two produces will have no node coincidence.

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 [6]. 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.

References

1. G.D. Battisa, P. Eades, R. Tamassia, and I.G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry: Theory and Applications*, 4:235–282, 1994.

2. A. Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):252–387, 1981.
3. M.J. Box, D. Davies, and W.H. Swann. *Non-linear optimization techniques.* Oliver & Boyd, 1969.
4. F. J. Brandenburg. Designing graph drawings by layout graph grammars. In *Proceedings of DIMACS International Workshop, GD'94, LNCS 894*, Princeton, New Jersey, USA, October 1994. Springer-Verlag.
5. F. J. Brandenburg, M. Himsolt, and C. Rohrer. An experimental comparison of force-directed and randomized graph drawing algorithms. In *Symposium on Graph Drawing, GD'95, LNCS 1027*, Passau, Germany, September 1995. Springer-Verlag.
6. S.S. Chok and K. Marriott. Automatic construction of user interfaces from constraint multiset grammars. In *IEEE Symposium on Visual Languages*, 1995.
7. I. F. Cruz and A. Garg. Drawing graphs by example efficiently: trees and planar acylic digraphs. In *Proceedings of DIMACS International Workshop, GD'94, Princeton, New Jersey, USA, October 1994, LNCS 894*, Princeton, New Jersey, USA, October 1994. Springer-Verlag.
8. R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. Technical report, Department of Applied Mathematics and Computer Science, 1991.
9. E. Dengler, M. Friedell, and J. Marks. Constraint-driven diagram layout. In *Proceedings of the 1993 IEEE Symposium on Visual Languages*, 1993.
10. P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.
11. P. Eades, W. Lai, K. Misue, and K. Sugiyama. Preserving the mental map of a diagram. In *Proceedings of Compugraphics '91*, pages 24–33, 1991.
12. P. Eades and J. Marks. Graph-drawing contest report. In *Symposium on Graph Drawing, GD'95, LNCS 1027*, Passau, Germany, September 1995. Springer-Verlag.
13. R. Fletcher. *Practical Methods of Optimization.* John Wiley & Sons, 1987.
14. A. Frick, A. Ludwig, and H. Mehldau. A fast adaptive layout algorithm for undirected graphs. In *Proceedings of DIMACS International Workshop, GD'94, LNCS 894*, Princeton, New Jersey, USA, October 1994. Springer-Verlag.
15. T. M J Fruchterman and E. M Reingold. Graph drawing by force-directed placement. *Software-Practice and Experience*, 21(11):1129–1164, November 1991.
16. A. Garg, M. T. Goodrich, and R. Tamassia. Area-efficient upward tree drawing. In *Proceedings of the 9th Annual Symposium on Computational Geometry, ACM*, 1994.
17. D Goldfarb and A Idnani. A numerically stable dual method for solving strictly convex quadratic programs. *Math. Prog.*, 27:1–33, 1983.
18. R. Helm and K. Marriott. Declarative graphics. In *Proc. of the 3rd International Conference on Logic Programming, LNCS 225*, pages 513–527, London, England, 1986. Springer-Verlag.
19. R. Helm and K. Marriott. A declarative specification and semantics for visual languages. *Journal of Visual Languages and Computing*, 2:311–331, 1991.
20. R. Helm, K. Marriott, T. Huynh, and J. Vlissides. An object-oriented architecture for constraint-based graphical editing. In *Object-Oriented Programming for Graphics*, pages 217–238. Springer-Verlag, 1995.
21. J. Q. Walker II. A node-position algorithm for general tree. *Software-Practice and Experience*, 20(7):685–705, July 1990.

22. T. Kamada. *Visualizing abstract objects and relations:a constraints-based approach*, volume 5 of *Computer Science*. Singapore, New Jersey:World Scientific, 1989.
23. T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, April 1989.
24. T. Kamps, J. Kleinz, and J. Read. Constraint-based spring-model algorithm for graph layout. In *Symposium on Graph Drawing, GD'95, LNCS 1027*, Passsu, Germany, September 1995. Springer-Verlag.
25. P. Kikusts and P. Rucevskis. Layout algorithm of graph-like diagrams for grade windows graphic editors. In *Symposium on Graph Drawing, GD'95, LNCS 1027*, Passau, Germany, September 1995. Springer-Verlag.
26. T. Lin and P. Eades. 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, 1994.
27. P. Luders, R. Ernst, and S. Stille. An approach to automatic display layout using combinatorial optimization. *Software-Practice and Experience*, 25(11):1183–1202, 1995.
28. K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages and Computing*, 6:183–210, 1995.
29. B. A. Myers, D. A. Giuse, R. B. Dannenberg, B. V. Zanden, D. S. Kosbie, E. Pervin, A. Mickish, and P. Marchal. Garnet: comprehensive support for graphical highly interactive user interfaces. *Computer*, pages 71–85, November 1990.
30. S. C. North. Incremental layout in dynadag. In *Symposium on Graph Drawing, GD'95, LNCS 1027*, Passau, Germany, September 1995. Springer-Verlag.
31. T. Lin P. Eades and X. Lin. Two tree drawing conventions. Technical Report 174, Key Centre for Software Technology, Department of Computer Science, The University of Queensland, 1990.
32. F. N. Paulisch. *The design of an extendible graph editor.* LNCS 704, Springer-Verlag, 1993.
33. E. M. Reingold and J. S. Tilford. Tidier drawing of trees. *IEEE Trans. on Software Engineering*, SE-7(2):223–228, March 1981.
34. K. Sugiyama and K. Misue. A simple and unified method for drawing graphs: magnetic-spring algorithm. In *Proceedings of DIMACS International Workshop, GD'94, LNCS 894*, Princeton, New Jersey, USA, 1994. Springer-Verlag.
35. K. Tsuchida, Y. Adachi, Y. Oi, Y. Miyadera, and T. Yaku. Constraints and algorithm for drawing tree-structured diagrams. In *Proceedings of the International Workshop on Constraints for Graphics and Visualization, CGV '95*, Cassis, France, September 1995.
36. D. Tunkelang. A practical approach to drawing undirected graphs. Carnegie Mellon University, 1994.
37. J. G. Vaucher. Pretty-printing of trees. *Software-Practice and Experience*, 10:553–561, 1980.
38. L. Weitzman and K. Wittenburg. Relation grammars for interactive design. In *Proceedings of IEEE Visual Languages*, pages 4–11, 1993.
39. C. Wetherell and A. Shannon. Tidy drawing of trees. *IEEE Trans. on Software Engineering*, SE-5(5):514–520, September 1979.