, , 1–31 ()

Removing Node Overlapping in Graph Layout Using Constrained Optimization

KIM MARRIOTT marriott@csse.monash.edu,au
School of Computer Science & Software Engineering, Monash University, Clayton 3168, Australia

PETER STUCKEY pjs@cs.mu.oz.au
Department of Computer Science & Software Engineering, University of Melbourne, Parkville 3052, Australia

VINCENT TAM vtam@comp.nus.edu.sg
Department of Computer Science, National University of Singapore, Singapore 119260, Singapore

WEIQING HE whe@csse.monash.edu,au
School of Computer Science & Software Engineering, Monash University, Clayton 3168, Australia

Abstract. Although graph drawing has been extensively studied, little attention has been paid to the problem of node overlapping. The problem arises because almost all existing graph layout algorithms assume that nodes are points. In practice, however, nodes may be labelled and these labels may overlap. We propose four different approaches for removing node overlapping, all of which are based on constrained optimization techniques. The first is the simplest. It performs the minimal linear scaling which will remove node-overlapping. The second approach relies on formulating the node overlapping problem as a convex quadratic programming problem which can then be solved by any quadratic solver. The disadvantage is that since constraints must be linear the node overlapping constraints cannot be expressed directly but must be strengthened to obtain a linear constraint strong enough to ensure no node overlapping. The third and fourth approaches are based on local search methods. The third is an adaptation of the EGENET solver originally designed for solving general constraint satisfaction problems while the fourth approach is a form of Lagrangian multiplier method, a well-known optimization technique used in operations research. Both methods are able to handle the node overlapping constraints directly and thus may potentially find better solutions. Their disadvantage is that no efficient global optimization methods are available for such problems, and hence we must accept a local minimum. We compare all of the above methods on a series of layout adjustment problems.

Keywords: graph layout, constrained optimization, disjunctive constraints

1. Introduction

Graph drawing has been extensively studied over the last fifteen years. However, almost all research has dealt with *graph layout* in which the nodes are treated as points in the layout of graphs. Unfortunately, treating nodes as points is inadequate for many applications. For example, a textual label is frequently added to each node to explain some important information, as in many illustrative diagrams for engineering designs, the human body and the geography, and even a satisfactory layout for a point-based graph may lead to node overlapping when labels are considered. For this reason, we are interested in *layout adjustment* which takes an initial graph layout with the sizes of the nodes as inputs, and then modifies the original graph layout so that node overlapping is removed.

We primarily address the problem of layout adjustment in a dynamic context. This is useful in interactive applications such as graph display in which sub-graphs are enlarged and shrunk or node labels are changed. Here the aim is take an existing graph layout and remove node overlapping while preserving the user's *mental map* of the graph. Following, Misue *et al* [17] in order to minimize change to the mental map we preserve the graph's *orthogonal ordering*, that is to say, the relative ordering of the nodes in both the x and y direction, and attempt to place nodes as closely as possible to their original positions. We also consider layout adjustment in the context of a static graph. In this case we wish to minimize the area of the graph in the new layout. However, since we assume that some sophisticated graph layout algorithm has been used to give an initial layout for the graph, like the dynamic case we still try to preserve the initial layout while removing node overlapping.

In this paper we study four different proposals for performing graph layout adjustment. All approaches model the problem as a constrained optimization problem. An advantage of viewing layout adjustment as a constrained optimization problem is that we can also add constraints which capture the semantics of the diagram if it is not just a simple graph. For example, we can add constraints which specify that certain nodes must be on the boundary of the graph or which specify the relative placement of nodes.

Our first approach to layout adjustment is the simplest. It performs the minimum linear scaling in both the x and y directions which will remove node-overlapping. We give a simple and efficient algorithm based on dynamic programming for finding this minimum. The disadvantage is that uniform scaling means that nodes are often moved unnecessarily far, and so this approach may not lead to good layout.

The second approach is based on formulating the node overlapping problem as a convex quadratic programming problem which can then be solved using a quadratic solver. This approach is attractive because algorithms for convex quadratic optimization are well understood, and global optimization is possible in polynomial-time. The main disadvantage arises from the need to model node overlapping constraints using linear constraints. Unfortunately, the no node-label overlap constraint between nodes u and v is inherently disjunctive, i.e. u is sufficiently to the left of v or u is sufficiently above v or u is sufficiently to the right of v or u is sufficiently below v. This cannot be expressed directly but must be strengthened to obtain a linear constraint strong enough to ensure no node overlapping. Optimal layout with respect to this stronger linear constraint may be sup-optimal with respect to the original no node-label overlap constraints.

In contrast the third and fourth approaches allow the disjunctive node overlapping constraint to be expressed directly. Both approaches use local search methods [1, 2, 7, 6, 27]. A local search method starts with a current value for each variable, and by examining the local neighbourhood tries to move to a point which is closer to the optimum. Constraints are handled as penalties to the optimization function. The search proceeds until a local minimum is found. When the local minimum represents a solution (all constraints are satisfied), the algorithm terminates. Otherwise, it may penalize this local minimum to avoid visiting it again, and then continue the search until it finds a solution. Because for layout adjustment problems we have a starting position "close" to the global optimum, namely the initial layout, in many cases local search can efficiently find the optimum.

The third approach is a modification of the EGENET [11] solver (an extension of the GENET model [27]) to perform layout adjustment. Modifications are required to (a) handle a constraint optimization problem (as opposed to constraint satisfaction for which EGENET is designed) and (b) to handle real (floating point) variables, rather than variables with a discrete domain. Our work represents the first attempt, that we know of, to investigate how the EGENET solver can be used to solve constrained optimization problems involving *real numbers*[1].

Our fourth approach is an adaptation of the Lagrangian multiplier methods (LMMs) [13, 20, 26], a classical optimization technique used in operations research. Lagrangian multiplier methods are a general approach to constrained optimization which have been applied to many difficult problems successfully. Here we develop a specific Lagrangian algorithm where variables are treated in pairs corresponding to node positions. Because of the non-convexity of the constraints, we repeatedly optimize using the results of the previous optimization to give a new, possibly better, starting position for the search.

There has been little work on layout adjustment and mental map preservation. The most closely related papers are that of Eades *et al* [9] in which *mental map* was proposed for the first time and *orthogonal ordering, proximity relations and topology* were given as three criteria, and that of Misue *et al* [17], which provides a force-directed algorithm called the *force scan algorithm* to perform dynamic graph layout adjustment. We demonstrate that our methods, apart from the uniform scaling approach, give better layout than the force scan algorithm, although they are slower.

Other related work includes Lyons [15] who tries to improve the distribution of the nodes in the new layout according to some measures of distribution (*cluster busting*) while simultaneously trying to minimize the difference between the two layouts according to some measures of difference (anchored graph drawing). Eades *et al* [8] try to preserve the mental map in animated graph drawing by using a modified spring algorithm to provide smooth user transitions. Other related work includes: [19], [4], [14], [18] and [21]. A preliminary version of our second proposal, including a proof, appeared in [10]. For a comprehensive survey of graph drawing methods, see [3, 22].

Finally we mention some previous work [24, 25] on adapting the GENET model to solve constraint optimization problems [23]. Guided Local Search (GLS) [25] iteratively calls a local search procedure, based on the GENET variable updating scheme, to modify and minimize an augmented cost function until a predefined stopping condition is reached. GLS had been successful in solving Radio Link Frequency Assignment Problems.

This paper is organized as follows. In Section 2 we describe how to view the layout adjustment problem as a constrained optimization problem and introduce both a static and dynamic version of the problem. In Section 3 we describe the simplest approach to solving the layout adjustment problem, linear scaling. In Section 4 we show how to replace the disjunctive no-node overlapping constraints of the layout problem with linear approximations. This allows us to transform the layout adjustment problem to a quadratic programming problem, which can then be handled by any quadratic solver. In Section 5 we introduce the original EGENET model. Then, we discuss how to modify the EGENET model to handle both the static and dynamic layout problems as continuous constrained optimization problems. In Section 6 we briefly describe some basic concepts in multiplier methods, and then discuss how to adapt these concepts to derive a Lagrangian-based search procedure specialized for layout adjustment problems. We provide an empirical evaluation of our four proposals using a set of arbitrary graph layout problems investigating both their efficiency and effectiveness in Section 7. We also compare our approaches to the force-scan algorithm of Misue *et al.* Finally, we summarize and conclude our work in Section 8.

2. Layout Adjustment as Constrained Optimization

All our algorithms for layout adjustment are based on translating the problem into a constrained quadratic optimization problem of the form

$$\text{minimize } \phi \text{ with respect to } C$$

where ϕ is a quadratic expression and C is a (possibly non-linear) collection of constraints. All are in terms of variables representing the position of each node.

We assume that we are given

- A (possibly directed) graph $G = \langle V, E\rangle$ where $V = \{1, \ldots, n\}$ is the set of nodes in the graph, and $E \subseteq V \times V$ is the set of edges in the graph, that is to say, $\langle u, v\rangle \in E$ whenever there is a edge from node u to node v.

- A *node labelling* for G which consists of two vectors $\mathbf{w} = (w_1, \ldots, w_n)$ and $\mathbf{h} = (h_1, \ldots, h_n)$ which map each node v to the width (w_v) and height (h_v) of its label respectively.

- And an *initial layout* for the graph G, which consists of two vectors, $\mathbf{x}^0 = (x_1^0, \ldots, x_n^0)$ and $\mathbf{y}^0 = (y_1^0, \ldots, y_n^0)$ which map each node v to its x or y position respectively. That is to say, node v is placed at (x_v^0, y_v^0).

Intuitively, as shown in Figure 1, each node has a rectangular bounding box and the display area has origin at its lower left corner with x axis rightwards and y upwards.

The variables of the layout adjustment problem are the x and y coordinates of the nodes, which we shall represent using two vectors $\mathbf{x}$ and $\mathbf{y}$. The constraints on the problem for layout adjustment must ensure that there is no node overlapping in the resulting solution. These constraints (C^{no}) can be expressed as: for all $u, v \in V$,

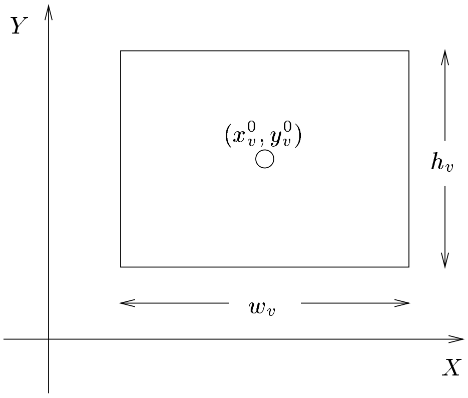


Figure 1. Notation

$u \neq v$ implies

$$\begin{array}{rrcll} & x_v - x_u & \geq & \frac{1}{2}(w_v + w_u) & (v \text{ to the right of } u) \\ \vee & x_u - x_v & \geq & \frac{1}{2}(w_v + w_u) & (u \text{ to the right of } v) \\ \vee & y_v - y_u & \geq & \frac{1}{2}(h_v + h_u) & (v \text{ above } v) \\ \vee & y_u - y_v & \geq & \frac{1}{2}(h_v + h_u) & (u \text{ above } v) \end{array}$$

or equivalently

$$|x_v - x_u| \geq \tfrac{1}{2}(w_v + w_u) \quad \vee \quad |y_v - y_u| \geq \tfrac{1}{2}(h_v + h_u).$$

In fact, we usually want the nodes to be separated by some minimum distance d and not directly abutt each other. This is simply handled by modifying the height and width of each node by adding the distance d, and treating the problem as no overlap of these new larger nodes.

As indicated in the introduction, we are primarily concerned with layout adjustment in the context of *dynamic* graph layout. This is useful in interactive applications such as graph display in which sub-graphs are enlarged and shrunk or node labels are changed. Here the aim is take an existing graph layout and remove node overlapping while preserving the user's *mental map* of the graph.

One heuristic to ensure that the aesthetic criteria remain satisfied and that the new layout is "similar" to the initial layout is to preserve the *orthogonal ordering* of the original layout [17]. The idea is to preserve the relative ordering of the nodes in the x and y directions. In our context, linear constraints (C^{oo}) which ensure the preservation of orthogonal ordering are: for each $u, v \in V$, if $x_u^0 < x_v^0$ then

$$x_u \ < \ x_v$$

and if $x_u^0 = x_v^0$ then

$$x_u \ = \ x_v$$

and similarly for the y-direction. We shall use this heuristic for our first two approaches—uniform scaling and quadratic programming—but not for the local search based methods.

The *dynamic layout adjustment problem* is to find a new layout for G, $\mathbf{x}$ and $\mathbf{y}$, so that the constraints C are satisfied and the position of each node is as close as possible to its original position. We encode the dynamic layout adjustment problem as a constrained optimization problem by setting the objective function ϕ_{dyn} to be,

$$\phi_{dyn} = \sum_{v \in V} (x_v - x_v^0)^2 + (y_v - y_v^0)^2,$$

and we wish to

$$\text{minimize } \phi_{dyn} \text{ subject to } C \tag{1}$$

where we can choose the constraints of the problem C to be either C^{no} or $C^{no} \wedge C^{oo}$. The new layout is given by an optimal solution to the above problem.

Apart from layout adjustment in a dynamic context, we might also consider layout adjustment in a static context. In this case the underlying assumption is that the initial layout has been generated using some sophisticated graph layout algorithm which captures aesthetic criteria. We do not want to redo this work in the layout adjustment phase. Hence, static layout adjustment should remove node-label overlapping but still preserve the (presumably) aesthetically pleasing node placement of the initial layout. We are also interested in minimizing the overall area of the graph. Preserving the initial layout is very similar to preserving the user's mental model, and so we can use the same techniques: constraints to preserve orthogonal ordering and a term in the objective function to move nodes as little as possible. The main difference to the dynamic layout adjustment problem is an extra term in the objective function, ϕ_{stat}, which allows the area of the new layout to be minimized.

More precisely, the *static layout adjustment problem* is to find a new layout for G, $\mathbf{x}$ and $\mathbf{y}$, by solving the constrained optimization problem

$$\text{minimize } \phi_{stat} + k \cdot \phi_{dyn} \text{ subject to } C \tag{2}$$

where C is either C^{no} or $C^{no} \wedge C^{oo}$, $k \geq 0$ is a weighting factor and

$$\phi_{stat} = \sum_{v \in V} (x_v - c^x)^2 + (y_v - c^y)^2$$

and $c = (c^x, c^y)$ is a "centre" towards which the layout is supposed to be shrunk. $c^x (c^y)$ can be the arithmetical average of all nodes' x (y) coordinates, or median of all nodes' x (y) coordinates. One can also take the position of centroid of a graph or even a predefined point as c. Minimization will attempt to place the nodes as close as possible to a predefined position, so reducing the overall area of the new layout.

Despite its different motivation, the formulation of the static layout adjustment problem is very similar to that for the static. Thus, in the following sections, we will focus on techniques for solving the dynamic case since they can also be used with little modification to solve the static case.

3. Using Uniform Scaling

We first consider a very simple approach to layout adjustment in which we uniformly scale the graph to remove overlapping. We find c_x, m_x, c_y and m_y and move each node (x_v^0, y_v^0) to $(m_x x_v^0 + c_x, m_y y_v^0 + c_y)$. Since this is a linear transformation, it preserves the graph's orthogonal ordering (indeed, all of the original graph's structure) and the scaling factors m_x and m_y are chosen so that node overlapping is removed. The disadvantage is that uniform scaling may cause nodes to move unnecessarily.

We first examine how to compute c_x and c_y for given scaling factors m_x and m_y. It follows from Equation 1 that we must minimize ϕ_{scale} where this is

$$\sum_{v \in V} (m_x x_v^0 + c_x - x_v^0)^2 + (m_y y_v^0 + c_y - y_v^0)^2 \tag{3}$$

This is minimized when

$$c_x = (1 - m_x)\mu_x \qquad\qquad c_y = (1 - m_y)\mu_y$$

where μ_x and μ_y are the means of the x_v^0's and y_v^0's, respectively. It follows that

$$\phi_{scale} = (m_x - 1)^2 \sigma_x^2 + (m_y - 1)^2 \sigma_y^2 \tag{4}$$

where σ_x^2 and σ_y^2 are the variances of the x_v^0's and y_v^0's, respectively.

Now we consider how to compute the scaling factors. Consider node (x_i, y_i) with dimensions (w_i, h_i) and node (x_j, y_j) with dimensions (w_j, h_j). They overlap if

$$\begin{array}{rrcl} & |x_i - x_j| & < & (w_i + w_j)/2 \\ \vee & |y_i - y_j| & < & (h_i + h_j)/2 \end{array}$$

Define

$$\begin{aligned} mx_{ij} &= \max\{(w_i + w_j)/(2|x_i - x_j|), 1\} \\ my_{ij} &= \max\{(h_i + h_j)/(2|y_i - y_j|), 1\} \end{aligned}$$

If $|x_i - x_j| = 0$ then $mx_{ij} = +\infty$ and similarly for my_{ij}. By construction mx_{ij} is the minimum amount of scaling in the x direction which will remove the overlap between the two nodes and similarly mx_{ij} is the minimum amount of scaling in the y direction to remove this overlap.

Hence we can solve the layout adjustment problem using scaling by finding scale factors sx and sy that solve the constrained optimization problem:

$$\text{minimize } \phi_{scale} \text{ subject to } \forall 1 \leq i < j \leq n (sx \geq mx_{ij} \vee sy \geq my_{ij}) \tag{5}$$

Inspection of Equation 4 reveals that we should choose the minimal scaling factors $m_x \geq 1$ and $m_y \geq 1$ that removes overlapping. This means that m_x will be mx_{ij} for some overlapping nodes i and j and similarly for m_y.

This observation leads to the algorithm shown in Figure 2. It computes c_x, m_x, c_y and m_y which remove all overlapping (if possible) and which minimize

```
layout-scale
    compute the mean mu_x and variance σ_x^2 of the x_v^0's
    compute the mean mu_y and variance σ_y^2 of the y_v^0's
    S := {(mx_ij, my_ij) | i, j ∈ V.i < j and i and j overlap}
    m := |S|
    sort S lexicographically into the array a[1...m]
    a[0] := (1, ∞)
    let a[k] have form (mx_k, my_k)
    /* compute possible pairs of scaling factors bmx[k] and bmy[k] */
    bmx[m] := mx_m
    bmy[m] := 1
    for k = m − 1 down to 0
        bmx[k] := mx_k
        bmy[k] := max{my_{k+1}, bmy[k + 1]}
    endfor
    /* now determine which pair has least cost */
    bestcost := ∞
    for k = 0 to m
        cost := (bmx[k] − 1)^2 σ_x^2 + (bmy[k] − 1)^2 σ_y^2
        if (cost < bestcost) then
            bestcost := cost
            best := k
        endif
    endfor
    if bestcost = ∞ then no solution
    else  m_x := bmx[best]
        m_y := bmy[best]
        c_x := (1 − m_x)μ_x
        c_y := (1 − m_y)μ_y
```

Figure 2. Scaling method for layout adjustment

ϕ_{scale}. The only subtlety to notice is that, for each i, the scaling factors $bmx[i]$ and $bmy[i]$ remove all overlapping since scaling by $bmx[i]$ in the x-dimension removes the overlapping corresponding to $a[1], ..., a[i]$ and scaling by $bmy[i]$ in the y-dimension removes the overlapping corresponding to $a[i+1], ..., a[m]$.

The main part of the algorithm has complexity $O(m \log m)$ where m is the number of overlapping nodes. Since the number of overlapping nodes is $O(|V|^2)$, the overall complexity is $O(|V|^2 \log |V|)$.

4. Using Quadratic Programming

Quadratic programming is used to find the global optimum of a convex quadratic objective function where the constraints are a conjunction of linear arithmetic equalities and inequalities. Quadratic programming has been widely studied in operations research and interior point methods provide polynomial time algorithms for solving such problems.

In Section 2 we saw how to encode the layout adjustment problem as a constrained optimization problem in which the objective function ϕ is a convex quadratic formula. Unfortunately, the constraints cannot be expressed as conjunctions of linear arithmetic constraints since the *no overlap* constraints involve disjunction.

To use a quadratic programming based approach, we need to replace these disjunctive constraints by linear approximations which will guarantee that the no overlap constraints hold. Since each individual disjunct is a linear constraint, the straightforward way to do this is to choose which disjunctive possibility must hold.

In order to construct the linear approximation of the no-overlap constraints C^{no} with respect to the x direction (C^{no}_x) we define the "right horizontal neighbours" of nodes. We then constrain each node to have no overlap with its right neighbours. In a sense we hardwire into the constraints which direction the nodes must move in to remove the overlapping.

For any node v, its *right horizontal neighbour nodes set* $right(v)$ is the set containing all nodes u such that in the initial position: (1) $u \neq v$; (2) u is to the right of v ($x^0_u > x^0_v$); and (3) u and v could overlap if moved only in the x direction (either $x^0_v - h_v/2 < x^0_u - h_u/2 < x^0_v + h_v/2$ or $x^0_v - h_v/2 < x^0_u + h_u/2 < x^0_v + h_v/2$ or $x^0_v - h_v/2 > x^0_u - h_u/2 \ \wedge x^0_u + h_u/2 > x^0_v + h_v/2$).

Node $r_v \in right(v)$ is an *immediate right horizontal neighbour node* of node v if there does not exist a node u such that: (1) $r_v \in right(u)$ and (2) $u \in right(v)$.

Similarly we can define the *upper vertical neighbour nodes set*, $upper(v)$, and *immediate upper vertical neighbour node*, u_v.

It is straightforward to define the constraints in C^{no}_x That is for each node $v \in V$, we compute its immediate right horizontal neighbours, and for each of these, r_v say, add the constraint

$$x_{r_v} - x_v \geq (w_{r_v} + w_v)/2$$

into C^{no}_x. Similarly we define C^{no}_y.

Generation of the orthogonal ordering constraints C^{oo}_x and C^{oo}_y for the x and y directions, respectively, is conceptually straightforward. For efficiency it is important to eliminate as many redundant constraints as possible. The precise algorithm is given in [10]

We treat the layout adjustment problem as two separate optimization problems, one for the x dimension and one for the y dimension, by breaking the optimization function into two parts and the constraint into two parts. The constraints in the x direction C_x are given by C^{no}_x together with the C^{oo} constraints on x variables (C^{oo}_x). We similarly define C_y.

For the dynamic layout adjustment problem it follows from the definition of ϕ_{dyn} that the optimization problems are to

$$minimize \qquad \phi_x = \sum_{v \in V} (x_v - x^0_v)^2 \ \ subject \ to \ C_x \tag{6}$$

quadratic-opt
compute C_x
$\mathbf{x}$:= minimize ϕ_x subject to C_x
$\mathbf{x}^0 := \mathbf{x}$
compute C_y
$\mathbf{y}$:= minimize ϕ_y subject to C_y

Figure 3. Quadratic programming approach to layout adjustment

for the x-direction, and

$$\textit{minimize} \quad \phi_y = \sum_{v \in V} (y_v - y_v^0)^2 \;\; \textit{subject to} \;\; C_y \tag{7}$$

for the y-direction. The new layout is given by $\mathbf{x}$ and $\mathbf{y}$ where $\mathbf{x}$ is the solution to (6) and $\mathbf{y}$ is the solution to (7).

The advantage in separating the problem in this way is twofold. First, it improves efficiency since it roughly halves the number of constraints considered in each problem. Second, if we solve for the x-direction first, it allows us to delay the computation of C_y to take into account the node overlapping which has been removed by the optimization in the x-direction.

The actual layout adjustment algorithm used is given in Figure 3. First the problem in the x dimension is solved. Then $\mathbf{x}^0$ is reset to be $\mathbf{x}$, the x positions discovered by this optimization. Doing this may reduce the number of upper vertical neighbours and so reduce the size of C_y and also allow more flexibility in node placement in the y-direction.

As we previously observed, polynomial time algorithms for solving quadratic programming problems exist. However, for medium sized problems of several hundred to one thousand constraints the preferred method of solution is an active set method. We have used an incremental implementation of the active set method provided by the C++ QOCA constraint solver [5]. The key idea behind the active set method is to solve a sequence of constrained optimization problems O_0, ..., O_t. Each problem minimizes f with respect to a set of equality constraints, $\mathcal{A}$, called the *active set*. The active set consists of the original equality constraints plus those inequality constraints that are "tight," in other words, those inequalities that are currently required to be satisfied as equalities. The other inequalities are ignored for the moment.

5. Using the EGENET Solver

In this section, we investigate how the EGENET solver can be used to solve the layout adjustment problem. In order to use quadratic programming we needed to linearize the no-overlap constraints and implicitly fixed choices about whether to remove the overlap in the vertical or the horizontal direction. This means that potentially better solutions to the layout adjustment problem were never considered. EGENET can handle the disjunctive no-overlap constraints directly and hence is attractive as a method for solving the constrained optimization problems associated with layout adjustment. But, because this formulation is non-convex only a local optimum can be found.

In the remainder of this section we first review the original EGENET model for solving discrete constrained satisfaction problems. We then examine how to modify the original EGENET to handle continuous constrained satisfaction problems (without optimization). Lastly, we discuss how the modified EGENET can be used to solve the layout adjustment problem as a continuous constrained optimization problem. To our knowledge, this work represents the first attempt to investigate how the EGENET solver can be used to solve continuous constrained optimization problems.

5.1. The Original EGENET Solver

A *constraint satisfaction problem* (CSP) [16] is a tuple (U, D, C), where U is a finite set of variables, D defines a finite set D_z, called the *domain* of z, for each $z \in U$, and C is a finite set of constraints restricting the combination of values that the variables can take. A *solution* is an assignment of values from the domains to their respective variables so that all constraints are satisfied simultaneously. CSPs are well-known to be NP-hard in general.

GENET [27] is an artificial neural network, based on the min-conflict heuristic (MCH), for solving arbitrary CSPs with binary constraints. The MCH is to assign a value causing the minimum number of constraint conflicts to each variable so as to quickly find the local minima in the search space. Lee *et. al.* [12] extended GENET to EGENET with a generic representation scheme for handling both binary and non-binary constraints. EGENET has been successfully applied to solve non-binary CSPs such as the car-sequencing problems and cryptarithmetic problems in an efficient manner [12].

Constraints in EGENET are represented by functions from values of the variables to non-negative numbers. For example the constraint $z_1 \geq z_2$ can be represented by the function

$$g(z_1, z_2) = \begin{cases} z_2 - z_1 & \text{if } z_2 > z_1 \\ 0 & \text{otherwise.} \end{cases}$$

Alternatively we can represent the same constraint just using a Boolean valued function, as

$$g(z_1, z_2) = \begin{cases} 1 & \text{if } z_2 > z_1 \\ 0 & \text{otherwise.} \end{cases}$$

A general CSP can be formulated as a a discrete unconstrained optimization problem as follows.

$$\min_{\mathbf{z} \in \mathbf{D}} f(\mathbf{z}) = \sum_{i=1}^{m} \lambda_i g_i(\mathbf{z})$$

where $\mathbf{D} = D_{z_1} \times \ldots \times D_{z_n}$ is the Cartesian product of the (finite) domains for all the n variables, m is the total number of constraints, and $g_i(\mathbf{z})$ denotes the penalty function for each constraint C_i in the CSP, and λ_i represents the weight given to the constraint. The penalty $g_i(\mathbf{z})$ equals 0 if the variable assignment $\mathbf{z}$ satisfies C_i, otherwise, it returns a positive integer. The weights associated with constraints are all initially 1, but may be modified by the EGENET procedure. In this formulation, the goal is to minimize the output of the cost function $f(\mathbf{z})$ whose value depends on the number of unsatisfied constraints and the weights associated with these constraints. For each solution $\mathbf{z}^*$ to the original CSP, $f(\mathbf{z}^*) = 0$ since $\mathbf{z}^*$ satisfies all the constraints.

EGENET uses a simple local search rule to minimize the cost function $f(\mathbf{z})$, and a heuristic learning rule to change weights of constraints until it finds a solution $\mathbf{z}^*$. Initially, a complete and random variable assignment $\mathbf{z}^0$ is generated. Then, the network executes a convergence procedure as follows. Each variable is asynchronously updated in each convergence cycle. The update simply finds the value for each variable which gives lowest total cost without modifying any other variables. When there is no change in any value assigned to the variables, the network is trapped in a local minima. If the local minimum does not represents a solution, a heuristic learning rule, is used to update the weight λ_i for any violated constraint C_i in the CSP so as to help the network escape from these local minima. The network convergence procedure iterates until a solution is found or a predetermined resource limit is exceeded.

5.2. *The Modified EGENET Solver*

The original EGENET solver only supports constraints over finite domains. Therefore, to solve the layout adjustment problems with the EGENET approach, we have to consider how to modify the original EGENET solver so as to handle problems with continuous domains.

To handle such a continuous constrained problem, we have to decide how to represent the real-number domain of each variable in an EGENET network. The domain of each variable in a CSP is finite and is usually represented by a finite set of contiguous integers. For continuous constrained problems, this is inappropriate, since, even if we consider that real variables only take floating point values, there are too many possibilities. Instead we represent the range of possible values of a variable z_i simply by a lower and upper bound $l_i..u_i$.

For problems with no natural lower and upper bounds on variables we need to generate them. The tighter the bounds used the more efficient the search will be since it examines a smaller area. But giving initial bounds which are too tight may lead to no solution being found even when one does exist.

Since the original EGENET variable updating function assumes a finite number of possible values in the domain of each variable, to update a continuous variable in the modified EGENET network, we need to treat the domain of each continuous variable represented as a range $l \ldots u$ as a set with finite number of elements bounded by the values l and u only. In other words, we *sample* the domain of a continuous variable by only trying a finite number of possible values in its range, when updating its value within the EGENET computation. The number of elements we try in each update is called the *domain sampling size*. The larger the domain sampling size the closer to the true local optimum solution we are likely to get, but the more computation is required at each step. We use a domain sampling size of 10 throughout our experiments, although this could be changed (even within a computation).

In order to overcome the coarseness of the search when using a small domain sampling size without increasing the computational overhead too greatly, whenever a solution is found we can revise the bounds of variables inwards towards the solution just found. This has the effect of focusing the search around the solution just found. This approach is used in the algorithm below. This extra flexibility makes our algorithm substantially different from the case where the domain sizes of the variables remain fixed throughout the computation (as used for discrete constrained optimization and satisfaction problems).

To handle a continuous constrained optimization problem we simply add a optimization component to the function to be minimized. The augmented cost function is

$$cost(\mathbf{z}) = \phi(\mathbf{z}) + f(\mathbf{z}).$$

As we shall see in the next section, this is closely related to the Lagrange multiplier methods.

Pseudo-code for the general optimization algorithm EGENET-opt is given in Figure 4. The algorithm takes the domain sampling size DSZ, the number n of variables in $\mathbf{z}$ and the cost function $cost()$ as input, and returns a tuple consisting of the best solution found $\mathbf{z}^b$ and its cost *best*. The basic idea is similar to that of GLS [25]. First, we initialize all the EGENET variables using initialize_vars (usually this is just random). Then, we use the var_order function to produce a permutation *perm* which determines the order in which the variables $\mathbf{z}$ will be updated. The core of the method is the updating loop where each of the sample values for a variable z_v are tried in turn, and the value which gives minimum cost is retained as the new value. This represents a simple local search on the best value of variable z_v. Because the cost function includes penalties for violated constraints this will drive the search towards solutions.

This loop continues until a local minima is found (where no variable is updated). When the local minimum represents a solution, EGENET-opt invokes the function revise_bounds to try to revise the bounds for the domains of the EGENET variables. Then, it computes the solution cost according to the cost function, and updates the best solution $\mathbf{x}^b$ and *best* if the cost of the current solution is smaller than *best*. To modify the augmented cost function, EGENET-opt invokes penalize_ctr($\mathbf{z}$), which is basically the same as the original EGENET heuristic learning mechanism, to penalize any violated constraint with respect to the current variable assignment. Similarly, EGENET-opt invokes penalize_opt($\mathbf{z}$) to modify the form of the optimization function that occurs in the augmented cost function. The algorithm iterates until the function stopping_criterion detects that a predefined stopping criterion is fulfilled, and therefore returns *true*.

```
EGENET-opt(DSZ, n, cost())
    initialize_vars(z)
    z^b := z
    best := cost(z^b)
    repeat
        repeat
            perm := var_order(n)
            for i = 1 to n
                v := perm[i]
                mincost := cost(z)
                minval := z_v;
                for j = 0 to DSZ − 1
                    z_v = l_v + j × (u_v − l_v)/DSZ
                    if cost(z) < mincost
                        mincost := cost(z)
                        minval := z_v;
                    endif
                endfor
                z_v := minval
            endfor
        until (no update for all v ∈ z)
        if (z represents a solution)
            revise_bounds(z)
            if (cost(z) < best)
                z^b := z
                best := cost(z^b)
            endif
        endif
        penalize_ctr(z)
        penalize_opt(z)
    until (stopping_criterion())
    return ⟨z^b, best⟩
```

Figure 4. A General EGENET-based Optimization Algorithm

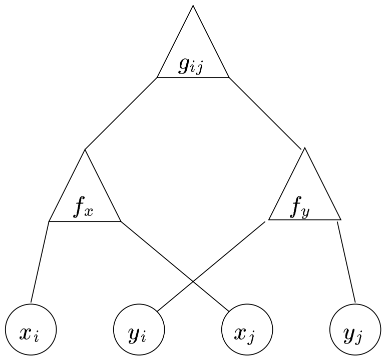
Clearly, the efficiency of using EGENET-opt to solve a continuous constrained optimization problem, and the quality[2] of the best solution found by EGENET-opt depends largely on how we define the augmented cost function, and how we appropriately modify this cost function by the penalize_ctr and penalize_opt function.

5.3. Handling Layout Adjustment

The variables of the layout adjustment problem are $\mathbf{x}$ and $\mathbf{y}$, the x and y coordinates of each node. Given the flexible representation scheme for any general constraint in the EGENET model, it is straightforward to define a *no-overlap* constraint as a disjunctive constraint in the EGENET network, which ensures that there will be no overlapping between the labels i and j, as follows:

$$\textit{no-overlap}(i,j) = |x_i - x_j| \geq (w_i + w_j)/2 \vee |y_i - y_j| \geq (h_i + h_j)/2$$

Figure 5 shows how the constraint is represented in the modified EGENET network. It defines a function g_{ij} which measures the amount of overlap in the x and y directions, returning 0 if there is no overlap. A no-overlap constraint is applied to each pair (i,j) of nodes where $1 \leq i < j \leq n$. This demonstrates one advantage of using the EGENET approach to solve the layout adjustment problems : it is, in general, simple to represent the constraints involved in these problems or other related graph layout problems in the EGENET network. Indeed arbitrary additional constraints can be added to the problem straightforwardly.



$$\begin{aligned} g_{ij} &= \max\{f_x, f_y\} \\ f_x &= \max\{0, (w_i + w_j)/2 - |x_i - x_j|\} \\ f_y &= \max\{0, (h_i + h_j)/2 - |y_i - y_j|\} \end{aligned}$$

Figure 5. The EGENET network for a no-overlap constraint.

We need to determine suitable ranges for the variables of the problem, in order to make use of the modified EGENET algorithm above. We would like to ensure that the variables' initial ranges contains a solution to the problem, if it exists. A conservative approach is to find a solution using the uniform scaling approach and then to use the minimal range containing that position and the original value. For instance, if (x_v^u, y_v^u) is the position computed by the uniform scaling approach for node v and the position of node v in the original graph is is (x_v^0, y_v^0) then the initial range for x_v can be set to $[x^0, x_u]$ if $x_u \geq x^0$ or $[x_u, x^0]$ otherwise, and similarly for y_v. Clearly this range is guaranteed to include at least one solution. The problem is that scaling factors tend to be large, so the range is also large.

In practice, by making some assumptions about the density of the nodes we can start with smaller ranges, and thus reduce the search. Define max_x to be the maximum overlap in the x direction of any pair of nodes, and define max_y similarly. We use initial ranges of $[x_v^0 - max_x..x_v^0 + max_x]$ for x_v and $[y_v^0 - max_y..y_v^0 + max_y]$ for y_v. This means no solution is guaranteed, but this is only true for graph where there is a dense overlapping in the initial layout. In practice this is rare.

For layout adjustment problems the initial value for the variables $\mathbf{x}$ and $\mathbf{y}$ returned by initialize_vars is given by the initial layout $\mathbf{x}^0$ and $\mathbf{y}^0$. One of the reasons for investigating local search methods is that this initial layout is usually quite close to the global optimum, and hence local search around the initial layout is likely to find a good solution.

The variable ordering strategy var_order used simply updates the variables in one dimension ($\mathbf{x}$) in a random order before or after updating the variables in the other dimension ($\mathbf{y}$). This is effective since the relationship between the $\mathbf{x}$ and $\mathbf{y}$ variables are weak (only through disjunctive constraints). By evaluating each dimension in turn we get a faster convergence to local minima. Other strategies are, of course, possible.

The revise_bounds function is defined as follows. When a solution is found at $\mathbf{x}$ then for each x_i, if $x_i > x_i^0$, we reset the upper bound to be the current value x_i and the lower bound is reset to x_i^0, if $x_i < x_i^0$ we reset the lower bound to be the current value x_i and the upper bound is reset to x_i^0, otherwise the bounds are unchanged. Similarly for $\mathbf{y}$. This encodes the strategy that, since we have found a solution on one side of the starting point, we will not look on the other side, and we will not look further away as looking farther away will tend to reduce the optimality of the solution.

For the graph layout problem, stopping_criterion returns *true* when a predetermined number of iterations have been tried, or when the cost of the current solution is worse than or equal to the previous solution. The total resource limit is initially set to 1000. Each time a better solution is found, the resource limit is reset to double the amount of resource used to find the previous solution if this is less than 100. Otherwise, the resource limit is reset to the minimum of 1000 or 120% of the resource used when finding the previous solution.

All that remains is to determine the penalize_ctr and penalize_opt functions. The relative weights of the objective function and the constraint penalties are important. For example, for the graph layout adjustment problem, if the augmented cost function is biased towards the penalty for constraint violation, that is node overlapping, then EGENET-opt may take a longer time, or even fail, to return a *good solution*.[3] On the other hand, if the augmented cost function is biased towards the solution cost, then EGENET-opt may fail to find a solution satisfying all the *no-overlap* constraints.

Recall the objective function ϕ_{dyn} for layout adjustment. As the quantity $(x_v - c_x)^2 + (y_v - c_y)^2$ (usually in the order of 100 or 1000) is always much larger than the penalty value of the constraints (which are Boolean) for constraint violations, we need to normalize this quantity so as to avoid bias towards the solution cost.

If the range of x_v is $[l_v..u_v]$ then the maximum value of the term $(x_v - c_x)^2$ is one of $(l_v - c_x)^2$ or $(u_v - c_x)^2$. Denote this by $omax_v$. The normalized optimization function for x_v is simply

$$norm_x(x_v) = (x_v - c_x)^2 / omax_v.$$

Clearly, the value returned by $norm_x(x_v)$ is a real number in the range of $[0 \dots 1]$. We can similarly define a normalized optimization function $norm_y$ for variables $\mathbf{y}$.

Accordingly, we define the augmented cost function $cost(\mathbf{x}, \mathbf{y})$ as as the sum of the penalty for constraint violations, and the sum of normalized optimization function as follows.

$$cost(\mathbf{x}, \mathbf{y}) = \sum_{v \in V} (norm_x(x_v) + norm_y(y_v)) + \sum_{j=1}^{m} \lambda_j g_j(\mathbf{x}, \mathbf{y})$$

where m denotes the total number of constraints in the problem.

When a local minimum is found, the penalty function penalize_opt changes the normalized optimization functions to take into account the new smaller ranges for some variables. The penalize_ctr function simply increases the weight λ_i of a violated constraint by 1 (as in the original EGENET model).

Note that in the EGENET approach we have ignored the orthogonal ordering constraints, and simply concentrated on the overlap constraints. Since, in practice, we have found that minimimizing ϕ_{dyn} preserves the structure of the original graph, and so tends to preserve orthogonality, even though there are not explicit constraints to do so.

6. Using A Pseudo-Lagrangian Method

The EGENET local search method described in the previous section is a modification of a discrete constrained satisfaction algorithm to solve continuous constrained optimization problems. In the end, it looks very similar to a Lagrange multiplier method. This inspired us to directly solve the problem using such an approach.

6.1. Lagrangian Multiplier Methods

Lagrange multiplier methods are a general approach to continuous constrained optimization that can tackle non-linear objective functions with non-linear constraints. A general continuous equality-constrained objective function is formulated as follows:

$$\text{minimize } f(\mathbf{z}) \text{ subject to } \mathbf{g}(\mathbf{z}) = 0 \tag{8}$$

where $f(\mathbf{z})$ is the objective function and $\mathbf{g}(\mathbf{z})$ is a vector of functions representing the constraint penalties.

The *Lagrangian function* associated with this problem is a weighted sum of the objective function and the constraints. It is defined as :

$$L(\mathbf{z}, \lambda) \;=\; f(\mathbf{z}) + \lambda^T \mathbf{g}(\mathbf{z}) \tag{9}$$

where λ is a vector of Lagrange multipliers. The Lagrangian function is related to the local extrema of the problem (8) by the following theorem (see e.g. [13]).

THEOREM 1 *Let* $\mathbf{z}$ *be a local extremum of* $f(\mathbf{z})$ *subject to* $\mathbf{g}(\mathbf{z})$*. Assume that* $\mathbf{z}$ *is a regular point*[4]*. Then there exists a vector* λ *such that*

$$\nabla_z f(\mathbf{z}) + \lambda^T \nabla_z \mathbf{g}(\mathbf{z}) = 0$$

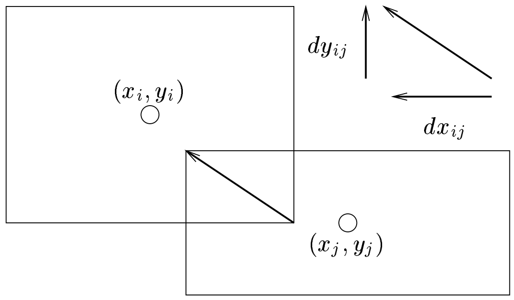


Figure 6. Overlap calculation.

Based on the above theorem there are a number of method for solving constrained optimization problems. The most widely used is the first-order method represented as an iterative process:

$$\mathbf{z}^{k+1} = \mathbf{z}^k - \alpha_k \nabla_z L(\mathbf{z}^k, \lambda^k)^T, \tag{10}$$
$$\lambda^{k+1} = \lambda^k + \alpha_k \mathbf{g}(\mathbf{z}^k) \tag{11}$$

where α_k is a step-size parameter. Intuitively the equations represent counteracting forces to achieve a good solution to the optimization problem. When a constraint is violated (11) increases the weight of the constraint, forcing the search towards a solution. In contrast (10) performs descent in the optimization direction once all of the Lagrange multipliers are fixed.

6.2. Layout Adjustment with Lagrange Multipliers

The Lagrangian approach to layout adjustment performs an iterative process like that defined above. Rather than a synchronous update of all variables at once, each node v is treated in turn, and its two coordinate variables x_v and y_v are updated together. This allows the non-overlap constraints to be handled in a more meaningful way.

When two nodes i and j overlap as illustrated in Figure 6, then the overlap of j on i in effect creates a force on i in the direction shown. If node i moves either a distance dx_{ij} in the x direction or dy_{ij} in the y direction the overlap will disappear. We choose the minimum magnitude of dx_{ij} or dy_{ij} as the resulting constraint violation. In effect, we treat the constraint function g_{ij} as follows:

$$dx_{ij} = \begin{cases} x_j - x_i + (w_i + w_j)/2 & \text{if } x_i - w_i/2 > x_j - w_j/2 \\ x_j - x_i - (w_i + w_j)/2 & \text{if } x_i + w_i/2 < x_j + w_j/2 \\ 0 & \text{otherwise} \end{cases}$$

$$dy_{ij} = \begin{cases} y_j - y_i + (h_i + h_j)/2 & \text{if } y_i - h_i/2 > y_j - h_j/2 \\ y_j - y_i - (h_i + h_j)/2 & \text{if } y_i + h_i/2 < y_j + h_j/2 \\ 0 & \text{otherwise} \end{cases}$$

$$g_{ij} = \begin{cases} (0,0) & \text{if } i \text{ and } j \text{ do not overlap} \\ (dx_{ij},0) & \text{if } dy_{ij} = 0 \vee dx_{ij} < dy_{ij} \\ (0,dy_{ij}) & \text{if } dx_{ij} = 0 \vee dy_{ij} < dx_{ij} \end{cases}$$

Given this definition then $\nabla_{x_i} g_{ij} = -1$ if $g_{ij} = (dx_{ij}, 0)$ and 0 otherwise, and similarly $\nabla_{y_i} g_{ij} = -1$ if $g_{ij} = (0, dy_{ij})$ and 0 otherwise

The two constraint functions g_{ij} and g_{ji} are symmetric and represent a single underlying overlap constraint. Hence, for each pair i, j where $i \neq j$ we have a Lagrange multiplier λ_{ij} representing the current weight of the constraint. The basic local search is then simply a Lagrangian optimization using a first-order stepping.

Since the optimization only finds a local minima, one optimization may not find a very good solution. After a local minima is reached the pseudo-Lagrangian method reduces the step-size by half, and also moves the nodes closer to their positions in the initial layout since this will further decrease the objective function albeit at the risk of introducing overlapping. This lets the search potentially find better solutions. Eventually after some number of such reoptimizations the method finishes, returning the best solution found. For the experiments the initial stepsize was 0.125 and the factor limit 255. The function initialize_multiplier simply initialized all the Lagrange multipliers to 1. The resource limit was never exceeded.

The algorithm in Figure 7 does not take into account the orthogonal ordering constraints. These could be added using the standard Lagrangian approach of adding new constraint functions, namely

$$g_{x_i = x_j} = x_i - x_j$$

and

$$g_{x_i < x_j} = \text{if } x_i \geq x_j \text{ then } x_j - x_i \text{ else } 0.$$

Unfortunately this simple addition can lead to divergent behaviour.

One simple approach to handle the equality constraints of the orthogonal ordering is instead to simply force their compliance. Let S be a set of coordinates which must all be equal, e.g. $\{x_1, x_3, x_6\}$. For each such set we compute the average $\bar{S} = (\sum_{x \in S} x)/|S|$ and set each value within to this average value, e.g. $x_1 := (x_1 + x_3 + x_6)/3$. This approach while seemingly ad-hoc, has quite a principled justification. If we replaced the set of variables S by a single variable (in effect using the equation constraints as substitutions) then the change in this variable would be determined from the sum of the changes of the variables in the set S. The average is simply a function of the sum of the changes of the variables in S. However, this approach does not handle the inequality constraints of the orthogonal ordering.

As for the EGENET approach, in our empirical evaluation of the Langrangian approach we have not included constraints to preserve the original orthogonal ordering, since again in practice, we have found that minimimizing ϕ_{dyn} preserves the structure of the original graph, and so tends to preserve orthogonality.

```
layout-mult(x^0, y^0, n, φ)
    x := x^0; y := y^0
    initialize_multiplier(λ)
    best := +∞
    stepsize := initial step size
    factor := 1;
    repeat
        for i = 1 to n
            dv_i := stepsize × ∇φ(x, y)
            for j = 1 to n
                if (i ≠ j)
                    dv_i += stepsize × λ_ij g_ij(x, y)
                    λ_ij += ||g_ij(x, y)||
                endif
            endfor
            x_i += dv_i(x); y_i += dv_i(y)
        endfor
        if (g(x, y) = 0)
            if φ(x, y) < best
                x^b := x; y^b := y
                best := φ(x, y)
            else
                factor ×= 2
                stepsize := stepsize/2
                x −= (x − x^0)/factor
                y −= (y − y^0)/factor
                initialize_multiplier(λ)
            endif
        endif
    until (factor > limit or resource limit exceeded)
    return ⟨x^b, y^b, best⟩
```

Figure 7. Pseudo-Lagrangian method for layout adjustment

7. Empirical Evaluation

In this section we compare the performance of the Scaling Algorithm (SCALE) presented in Section 3, the Force Scan Algorithm (FSA) of Misue *et al* [17], the quadratic solver approach (QUAD) presented in Section 4, the modified EGENET solver described in Section 5 and the Psuedo-Lagrangian method (PLM) discussed in Section 6 on a set of nine dynamic graph layout adjustment problems.

QUAD is implemented in *Borland C++ for Windows* Version 4.5. The other solvers are implemented in C, and compiled by the *GCC compiler* Version 2.7.3 on *Linux*. All tests were performed on a *Pentium PC* running at $155Mhz$.

Table 1. CPU time taken by SCALE, FSA, QUAD, EGENET and PLM for layout adjustment of the example problems.

Graph	# nodes	SCALE	FSA	QUAD	EGENET	PLM
1	14	0.010	0.020	0.110	4.959	0.413
2	6	0.000	0.000	0.000	0.094	0.027
3	25	0.030	0.020	0.110	1.116	0.069
4	10	0.010	0.000	0.060	0.037	0.011
5	10	0.000	0.010	0.000	0.064	0.019
6	5	0.000	0.000	0.000	0.035	0.008
7	12	0.000	0.020	0.110	19.564	0.210
8	9	0.010	0.010	0.060	1.115	0.064
9	17	0.000	0.010	0.050	1.302	0.245

Table 2. Value of ϕ_{dyn} in the adjusted layout using SCALE, FSA, QUAD, EGENET and PLM for layout adjustment on the example problems.

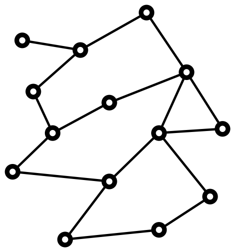
Graph	# nodes	SCALE	FSA	QUAD	EGENET	PLM
1	14	44135	204298	21023	22145	*7584
2	6	*450	1960	1028	594	874
3	25	40928	29160	*9552	4640	5544
4	10	16177	6272	* 393	784	424
5	10	1027	900	657	450	*162
6	5	699866	132930	51103	*5700	3012
7	12	750800	2589242	210846	93294	*65023
8	9	160136	315446	64362	20564	*11792
9	17	38880	127008	*77760	5535	3688

A quantitative comparison of these different methods on the sample problems is provided in Table 1 and 2. In each table, the first column gives the identifying number for the graph while the second column gives the number of nodes in the sample graph. Table 1 details the CPU time in seconds taken to find the adjusted layout for each method while Table 2 details the value of ϕ_{dyn} for the adjusted layout given by each method.

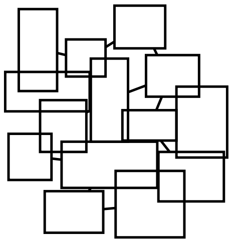
For local search methods such as EGENET and PLM, the averages of CPU time and cost over 10 successful runs are reported in each case. For all the cases we have tested, both EGENET and PLM can successfully find an (sub-)optimal final layout without node overlapping. It should be noted that for both EGENET and PLM, the solvers will halt when the current solution found is worse than the previous one. For EGENET, the solver will also halt after the, possibly reset, resource limit is exceeded.

Broadly speaking we find that the ranking of methods with respect to the CPU time taken to find the adjusted layout, from fastest to slowest, is SCALE, FSA, QUAD, PLM and EGENET. SCALE and FSA are considerably faster than the other methods, QUAD somewhat faster than PLM and the modified EGENET solver takes much longer than the other solvers. This is probably because the modified EGENET solver can only consider a finite number of points in the domains of each variable for each updating, and slowly revise the domains after each learning.

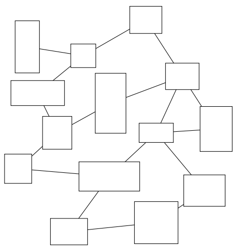
The values of ϕ_{dyn} reported in Table 2 provide a simple numerical measure of the quality of the adjusted layout. However, it is also important to look at the actual aesthetic quality of the adjusted layout. For this reason we now look at each of the example graphs and show the resulting layout adjustment with each method. Note that we have used an asterisk (*) to indicate the method giving the (subjectively) best layout for each example in Table 2.



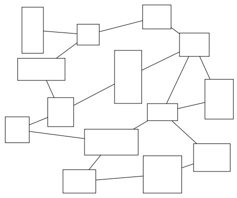
(a) Graph 1, general



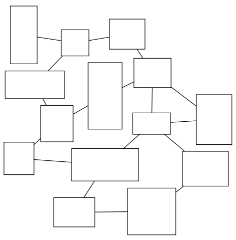
(b) Graph 1 with box nodes



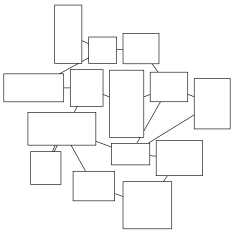
(c) Layout with SCALE



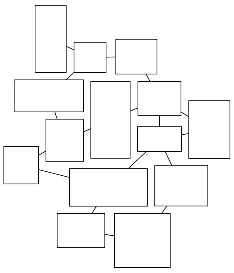
(d) Layout with FSA



(e) Layout with QUAD



(f) Layout with EGENET

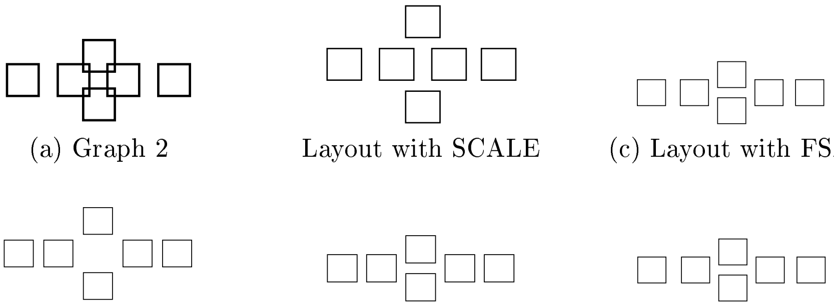


(g) Layout with PLM

Figure 8. Initial and layout adjustment for Graph 1.

Figure 8(a) and 8(b) respectively show the initial layout of Graph 1 as an idealized graph with circles as nodes and as a labelled graph. Figure 8(c), 8(d), 8(e), 8(f) and 8(g) give the resulting layout adjustment using SCALE, FSA, QUAD, modified EGENET and PLM respectively. All methods provide reasonable layout adjustment, although we note that EGENET introduces an edge/node label overlap. PLM gives the best adjustment.

Note that, as in all of our figures, we have scaled each graph to have a maximum height or width of one inch. Thus the adjusted layouts may not have the same scale. Smaller (usually better) layouts may be identified by their relatively larger node labels.



(a) Graph 2 Layout with SCALE (c) Layout with FSA

(d) Layout with QUAD (e) Layout with EGENET (f) Layout with PLM

Figure 9. Initial and resulting layouts for Graph 2.

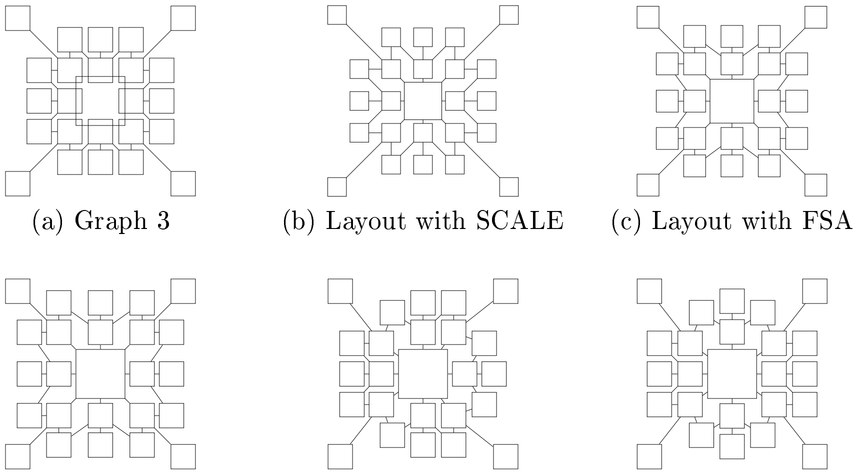
Surprisingly for Graph 2 as shown in Figure 9, the simplest method SCALE finds the best layout, while the other methods only find a local minimum. All methods give reasonable layout adjustment with QUAD giving the worst layout.

Figure 10 shows the initial and resulting layouts for Graph 3 which has been chosen as an example of the kind of explanation diagram widely used in Biology or Engineering textbooks. Arguably, QUAD gives the best adjusted layout since it best preserves the original graph's structure. SCALE increases the size by too much while EGENET and to a lesser extent PLM change the orthogonal ordering of the graph and remove its symmetry.

Graph 4 as shown in Figure 11(a) is an example of a rooted tree. Such labelled graphs are commonly used to display data structures or organization structures in many real-life applications. All methods give reasonable layout. The worst is by SCALE since it unnecessarily increases the size, while the layout found by QUAD is slightly better than that found by PLM and EGENET.

Graph 5 is another example of rooted tree layout. This time PLM gives the best layout, although it has changed the orthogonal ordering, closely followed by FSA which preserves the orthogonal ordering. Both QUAD and EGENET introduce an edge/label crossing.

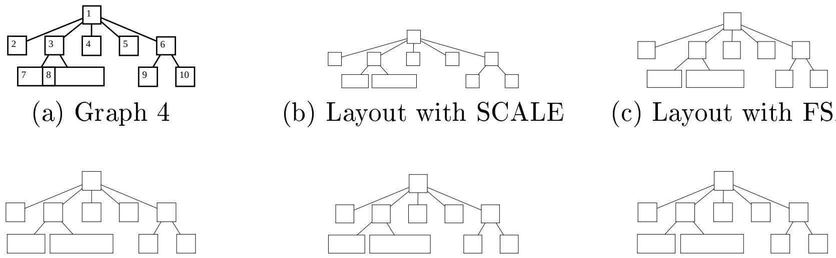
Graph 6 was carefully designed to show that layout adjustment may introduce edge crossings even though the layout adjustment method preserves the graph's orthogonal ordering. Both FSA and QUAD produce the worst layouts since they introduce edge crossing. SCALE does not introduce any edge crossing in the final



(a) Graph 3 (b) Layout with SCALE (c) Layout with FSA

(d) Layout with QUAD (e) Layout with EGENET (f) Layout with PLM

Figure 10. Initial and resulting layouts for Graph 3.



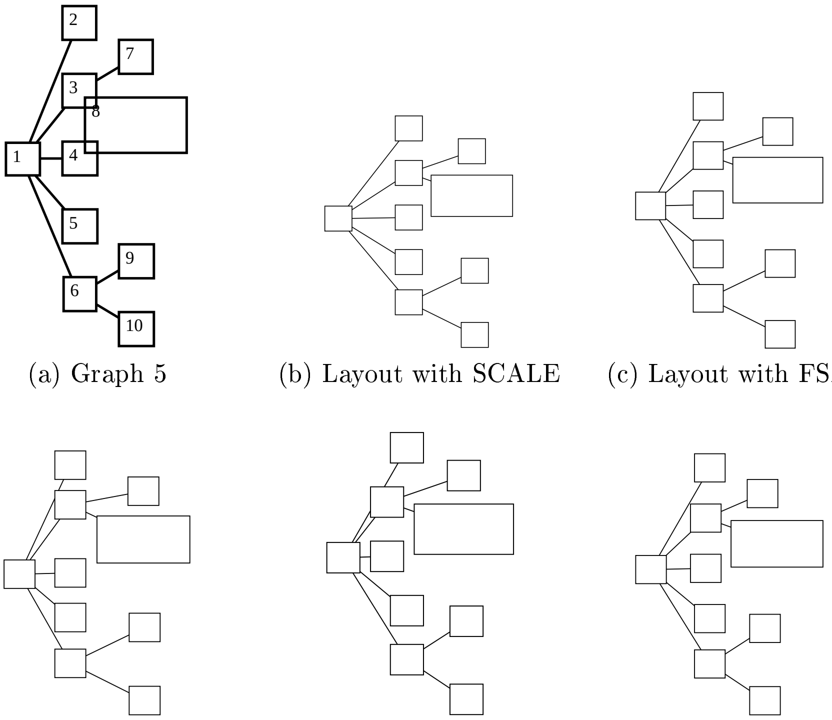
(a) Graph 4 (b) Layout with SCALE (c) Layout with FSA

(d) Layout with QUAD (e) Layout with EGENET (f) Layout with PLM

Figure 11. Graph 4 — tree layout adjustment (1).

layout since the initial layout does not contain any edge crossing, but the resulting layout is, as usual, unnecessarily wide. The modified EGENET and PLM methods produce very similar (and good) layouts with that produced by EGENET slightly better than that produced by PLM.

As shown in Figure 14(a), Graph 7 is a rather pathological graph with no edges but with lots of node overlapping occurred at different horizontal levels. SCALE clearly gives the worst layout adjustment while the modified EGENET gives the best layout which is neatly packed and fairly close to the initial layout. On the other hand, PLM gives a stack-like layout which looks quite different from the original graph.



(a) Graph 5 (b) Layout with SCALE (c) Layout with FSA

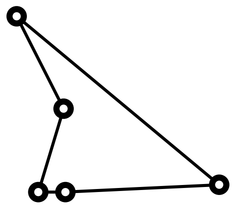
(d) Layout with QUAD (e) Layout with EGENET (f) Layout with PLM

Figure 12. Graph 5 — tree layout adjustment (2).

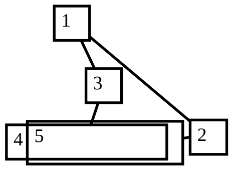
Graph 8 shown in Figure 15(a) is a simplified version of Graph 7 with some nodes removed. Thus, there is less node overlapping in Graph 8. All of the algorithms produce a result closer to the original graph, but in this case PLM gives the most aesthetically pleasing layout.

Graph 9 is an X-shaped graph with symmetry about both the x- and y-axis. All layouts are reasonable. The layouts by SCALE, FSA and QUAD retain the symmetry while those produced by the modified EGENET and PLM do not. QUAD produces the best layout.

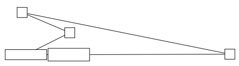
As we can see no method produces uniformly better layout adjustment, even SCALE produces the best layout for one example. However in general, SCALE and FSA produce the worst layout. In general, PLM closely followed by QUAD produce the best layout. However, EGENET and PLM can lose the original structure of the graph and to preserve this structure we would need to add extra constraints into the modified EGENET and PLM solvers. Generally, QUAD and FSA preserve the structure, in part because they preserve the orthogonal ordering, but may still introduce edge overlapping not found in the original graph. SCALE is guaranteed to preserve all of the original structure since it performs a simple uniform scaling.



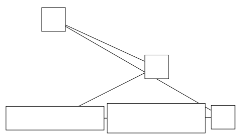
(a) Graph 6 without node labels



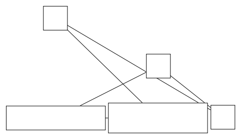
(b) Graph 6 with node labels



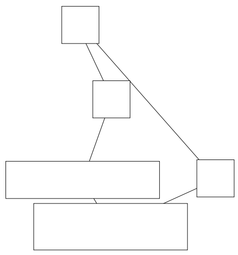
(c) Layout with SCALE



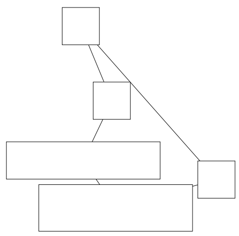
(d) Layout with FSA



(e) Layout with QUAD



(f) Layout with EGENET



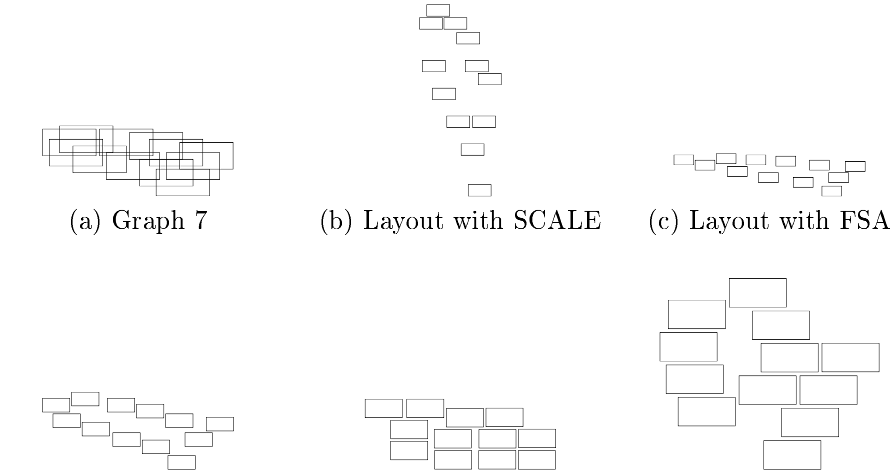
(g) Layout with PLM

Figure 13. Initial and resulting layouts for Graph 6.

7.1. Resource-bounded Layout Adjustment

Although the PLM solver produces the best layout, Table 1 suggests that it is substantially slower than the SCALE and FSA solvers. A natural question to ask is, given a (small) fixed amount of time, which method will give the best layout? This question makes sense because the PLM solver employs local search techniques and at any point in time has a current best solution. Thus we can stop the PLM solver after any time interval and look at the quality of the solution and compare this to that of the SCALE and FSA solvers. It does not make sense to perform the same experiment with QUAD since there is no concept of "the current best solution."

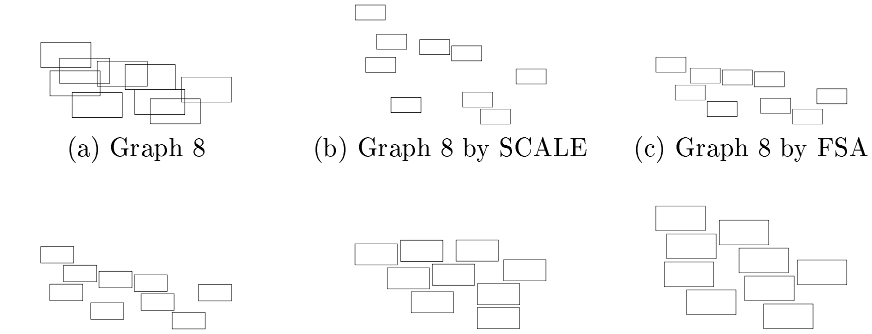
The graph in Figure 17 shows the cost of the best solution found so far (as a multiple of the best found eventually) versus time for the graphs 1, 7 and 9 during the execution of the PLM algorithm. The first non-overlapping solution is found for each graph within 0.02 seconds. Except for Graph 9 the value of ϕ_{dyn} for this solution is smaller than that of the solution eventually found by SCALE, FSA or



(a) Graph 7 (b) Layout with SCALE (c) Layout with FSA

(d) Layout with QUAD (e) Layout with EGENET (f) Layout with PLM

Figure 14. Initial and resulting layouts for Graph 7.



(a) Graph 8 (b) Graph 8 by SCALE (c) Graph 8 by FSA

(d) Graph 8 by QUAD (e) Graph 8 by EGENET (f) Graph 8 by PLM

Figure 15. Initial and resulting layouts for Graph 8.

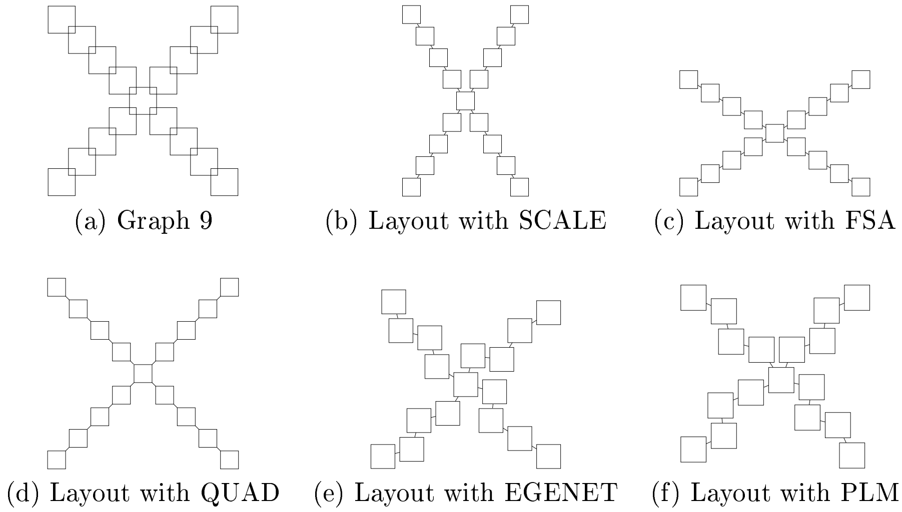


Figure 16. Initial and resulting layouts for Graph 9.

QUAD (for Graph 9 the first solution has cost 8108). For each of the graphs PLM finds a solution whose value of ϕ_{dyn} is within 20% of the eventual best in less than half the time required to find the best. For these examples we could safely stop PLM after 0.15 seconds and obtain a solution within 20% of the best found and, in fact, better than any solution found by the other algorithms.

8. Conclusion

We have studied the problem of layout adjustment for graphs in which we wish to remove node overlapping while preserving the graph's original structure and hence the user's mental map of the graph.

We have given four algorithms to solve this problem, all of which rely on viewing it as a constrained optimization problem. Empirical evaluation of our algorithms shown that they are reasonable fast and give nice layout, which is better than that of the comparable algorithm, the FSA of Misue *et al.* Generally speaking, the approach based on Lagrangian methods gives the best layout in a reasonable time. However, it may not preserve the orthogonal layout of the original graph. The quadratic programming approach also produces good layout in a reasonable time and does preserve the graph's orthogonal ordering. Uniform scaling gives the fastest and simplest approach to layout adjustment. It may lead to unnecessary enlargement of the graph, but is guaranteed to preserve all of the original structure in the graph.

Our results are not only interesting for layout adjustment of graphs: They also suggest techniques for laying out non-overlapping windows and labels in maps.

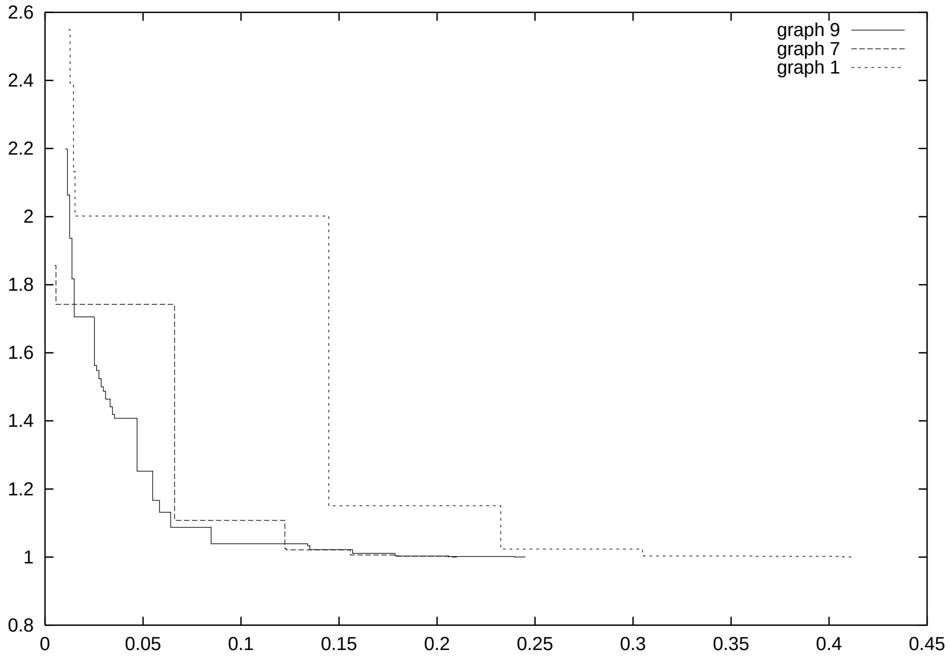


Figure 17. Cost of best solution found so far against time for graphs 1, 7, and 9

9. Acknowledgement

We would like to thank Peter Eades for his comments on our work, and Yi Xiao for the quadratic solver.

Notes

1. Note that unless the number of pixels is used, real numbers are usually used to denote the positions and sizes of the nodes in a graph layout since it allows greater flexibility.
2. Even though the global optimality of the resulting solution cannot be guaranteed by the EGENET approach as a local search method, the experimental results of the related GLS approach reported in [25] show that for a set of the real-life military frequency assignment problems GLS always found better solutions than those found by conventional search methods.
3. Here, we mean a solution with its cost close enough to the globally optimal cost of the optimization problem.
4. A *regular point* of constraints $\mathbf{g}$ is one where $\nabla g_1(\mathbf{z}), \ldots, \nabla g_m(\mathbf{z})$ are linearly independent.

References

1. E.H.L. Aarts and J.H.M. Korst. Boltzmann machines for traveling salesman problems. *European Journal of Operational Research*, 39:79–95, 1989.

2. H. Adorf and M. Johnston. A discrete stochastic neural network algorithm for constraint satisfaction problems. In *Proceedings of the International Joint Conference on Neural Networks*, 1990.
3. G. Di. 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.
4. K. Böhringer and F. N. Paulisch. Using constraints to achieve stability in automatic graph layout algorithms. In *CHI'90 Proceedings*, pages 43–51. ACM, 1990.
5. A. Borning, K. Marriott, P. Stuckey, and Y. Xiao. Solving linear arithmetic constraints for user interface applications. In *Proceedings of the 1997 ACM Symposium on User Interface Software and Technology*, pages 87–96, October 1997.
6. Andrew Davenport, E.P.K. Tsang, C.J. Wang, and K. Zhu. GENET: A connectionist architecture for solving constraint satisfaction problems by iterative improvement. In *Proceedings of AAAI'94*, 1994.
7. G. Dozier, J. Bowen, and D. Bahler. Solving small and large scale constraint satisfaction problems using a heuristic-based microgenetic algorithm. In *Proceedings of the IEEE International Conference on Evolutionary Computation*, 1994.
8. P. Eades, M. L. Huang, and J. Wang. Online animated graph drawing using a modified spring algorithm. In *ACSC '98 Proceedings*. Springer-Verlag, 1998. To appear.
9. P. Eades, W. Lai, K. Misue, and K. Sugiyama. Preserving the mental map of a diagram. Technical Report IIAS-RR-91-16E, International Institute for Advanced Study of Social Information Science, Fujitsu Laboratories Ltd., 1991.
10. W. He and K. Marriott. Removing node overlapping using constrained optimization. Technical report, Department of Computer Science, Monash University, 1997.
11. J.H.M. Lee, H.F. Leung, and H.W. Won. Extending GENET for Non-Binary CSP's. In *Proceedings of Seventh International Conference on Tools with Artificial Intelligence*, 1995.
12. J.H.M. Lee, H.F. Leung, and H.W. Won. Towards a more efficient stochastic constraint solver. In *Proceedings of Principles and Practice of Constraint Programming (CP96)*, 1996.
13. D.G. Leunberger. *Linear and Nonlinear Programming.* Addison-Wesley Publishing Company, 1984.
14. P. Lüders, R. Ernst, and S. Stille. An approach to automatic display layout using combinatorial optimization. *Software-Practice and Experience*, 25(11):1183–1202, 1995.
15. K. A. Lyons. *Cluster busting in anchored graph drawing.* PhD thesis, Department of Computing & Infromation Sci., Queen's Univ., 1996.
16. A. K. Mackworth. Consistency in networks of relations. *AI Journal*, 8(1):99–118, 1977.
17. 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.
18. A. Papakostas, J. M. Six, and I. G. Tollis. 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, 1997. New York:Springer.
19. F. N. Paulisch and W. F. Tichy. Edge: an extendible graph editor. *Software – Practice and Experience*, 20(S1):63–88, 1990.
20. D.M. Simmons. *Nonlinear Programming for Operations Research.* Prentice-Hall, Englewood Cliffs, NJ, 1975.
21. M. D. Storey and H. A. Müller. Graph layout adjustment strategies. In *Symposium on Graph Drawing, GD'95, LNCS 1027*, pages 487–499, Passau, Germany, September 1995. Springer-Verlag.
22. R. Tamassia, G. Di. Battista, and C. Batini. Automatic graph drawing and readablity of diagrams. *IEEE Transaction on Systems, Man, and Cybernetics*, 18(1):61–79, 1988.
23. Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.
24. C. Voudouris and E. Tsang. The tunneling algorithm for partial csps and combinatorial optimization problems. Technical Report CSM-213, Dept. of Computer Science, University of Essex, 1994.
25. C. Voudouris and E. Tsang. Partial constraint satisfaction problems and guided local search. Technical Report CSM-250, Dept. of Computer Science, University of Essex, 1995.
26. G.R. Walsh. *Methods of Optimization.* John Wiley and Sons, 1975.

27. C. Wang and E. Tsang. Solving satisfaction problems using neural-networks. In *Proceedings of IEE Second International Conference on Artificial Neural Networks*, 1991.