IPSEP-COLA: An Incremental Procedure for Separation Constraint Layout of Graphs

Tim Dwyer, Yehuda Koren, Member, IEEE, and Kim Marriott

Abstract— We extend the popular force-directed approach to network (or graph) layout to allow *separation constraints*, which enforce a minimum horizontal or vertical separation between selected pairs of nodes. This simple class of linear constraints is expressive enough to satisfy a wide variety of application-specific layout requirements, including: layout of directed graphs to better show flow; layout with non-overlapping node labels; and layout of graphs with grouped nodes (called clusters). In the stress majorization force-directed layout process, separation constraints can be treated as a *quadratic programming* problem. We give an incremental algorithm based on *gradient projection* for efficiently solving this problem. The algorithm is considerably faster than using generic constraint optimization techniques and is comparable in speed to unconstrained stress majorization. We demonstrate the utility of our technique with sample data from a number of practical applications including gene-activation networks, terrorist networks and visualization of high-dimensional data.

Index Terms—Graph drawing, constraints, stress majorization, force directed algorithms,multidimensional scaling.

Many fields of science, technology and industry require visualization of networks. For example, biologists study gene-activation networks and metabolic pathways, police study networks of associations between suspects to uncover organized crime or terrorist cells, and software and process engineers need to understand the complex networks of relationships between system components.

A wide variety of graph layout algorithms have been developed to aid such visualization. However, many of these algorithms are designed to draw simple, idealized mathematical graphs. This significantly limits their usefulness since, in many applications, the networks have much more complex structure and, consequently, more constraints on their layout. Such constraints include, for instance, requiring directed connections to be represented by arrows that point downward, grouping of selected nodes into clusters, large labels on nodes and edges, alignment of selected nodes, and an ordering on nodes perhaps reflecting an underlying physical ordering.

Current techniques for handling these application specific layout requirements are complex and are brittle in the sense that each technique can only handle a particular kind of layout constraint. Here we present IPSEP-COLA, an Incremental Procedure for Separation Constraint Layout of graphs. This is a new approach to network layout that provides a generic, robust framework that handles the constraints arising in a wide variety of applications. Our approach is relatively simple and is efficient enough to handle large networks with thousands of nodes.

The key idea behind IPSEP-COLA is to extend force-directed placement approaches for graph layout to allow so-called *separation constraints*. Force-directed placement algorithms are among the most successful approaches to the layout of simple graphs. They find an embedding of the graph in 2-D (or 3-D) space that minimizes some continuous goal function. A popular algorithm in this family has been that of Kamada and Kawai [16] which attempts to minimize the sum of squared differences between ideal spacing for pairs of nodes and their Euclidean distance in the embedding. These approaches are relatively easy to implement, they can — with appropriate hierarchical data structures — scale up to large graphs, and they give reasonable layout for most input graphs. They are the most commonly used technique for drawing unstructured networks. The starting point for our approach is Gansner et al. [11] who recently revisited force-directed placement and suggested using *functional* or *stress majorization* — an optimization technique from the field of multidimensional scaling. Compared to Kamada and Kawai, stress majorization has been shown to be faster and exhibit more robust convergence behavior.

We extend stress majorization to allow so-called *separation constraints* in each dimension. These have the form $u + d \leq v$ where u and v are variables representing horizontal or vertical position and d is a constant giving the minimum separation required between u and v.[1] Although seemingly a very restricted kind of linear constraint, separation constraints are expressive enough to handle a wide variety of application-specific layout constraints. These include:

Directed edges We can ensure that node v is placed above (or, to the left of) node u if there is a directed edge from v to u.

Alignment or distribution E.g. placing selected nodes on different horizontal layers.

Bands By adding dummy variables we can ensure that nodes are placed in vertical or horizontal bands, as defined in [3].

Fixed position A node's position can be fixed in any axis.

Containment We can ensure that selected nodes lie in a rectangular region, for instance within the boundaries of page, window or a cluster dynamically sized to fit its contents.

Orthogonal ordering between nodes We can ensure that nodes are to the left/right or above/below other nodes.

Non-overlap of nodes and/or clusters By dynamically generating separation constraints we can ensure that nodes do not overlap each other and, in combination with containment constraints, that clusters do not overlap.

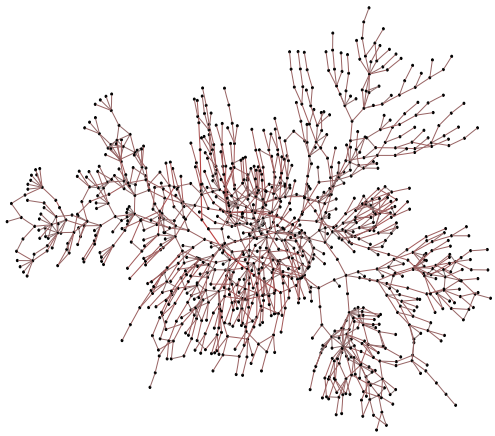
Our approach also handles dynamic layout. When a graph is modified in an interactive context usually we wish to find a new layout which is similar to the old layout, so as to preserve the viewer's "mental map" of the graph [17]. It is straightforward to add separation constraints to preserve the orthogonal ordering of the nodes in the layout.

Extending stress majorization to handle separation constraints has broader application in data visualization than just network layout since it allows this widely used multi-dimensional scaling technique to take into account layout constraints from the underlying structure. For instance placing data points in clusters or preserving some underlying ordering.

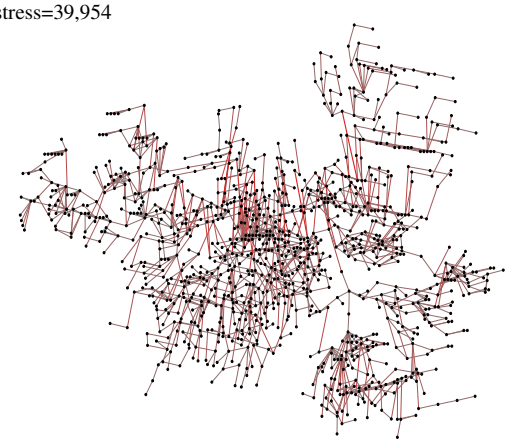
[1] In addition we can also handle equalities of the form $u + d = v$.

- *Tim Dwyer is with Monash University, Australia, E-mail: Tim.Dwyer@infotech.monash.edu.au.*
- *Yehuda Koren is with AT&T Labs — Research, E-mail: yehuda@research.att.com.*
- *Kim Marriott is with Monash University, Australia, E-mail: Kim.Marriott@infotech.monash.edu.au.*

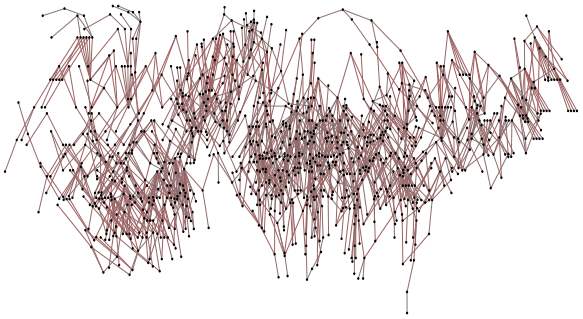
Manuscript received 31 March 2006; accepted 1 August 2006; posted online 6 November 2006.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.



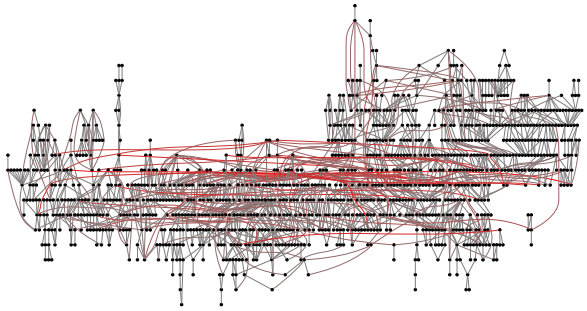
(a) Unconstrained layout – no uniform edge direction, 1,142 edge crossings, stress=39,954



(b) Drawn as a DAG using a separation constraint for every directed edge – all edges point downwards, 3,617 edge crossings,stress=49,035



(c) Drawn as a DAG using DiG-CoLa style level constraints – all edges point downwards and nodes are globally ordered according to hierarchical levels, 7,600 edge crossings, stress=74,894



(d) Drawn as a DAG using Sugiyama style layout, nodes arranged into levels such that all edges point downwards, 6,148 edge crossings, stress=N/A

Fig. 1. Drawings of the bus1138 graph [1] using different sets of constraints. Stress is computed using Eq. (1) with a desired edge length of 1. Long edges are highlighted in red.

This paper has two main contributions. The first is identifying the usefulness of separation constraints for modeling a wide variety of application specific network layout requirements. The second is an efficient algorithm for solving the force-directed placement problem in the presence of separation constraints. Stress majorization iteratively improves the drawing by minimizing a sequence of quadratic goal functions. We modify the method to minimize each quadratic goal function while satisfying the separation constraints. We solve the resulting Quadratic Program (QP) using an iterative *gradient-projection* algorithm that relies on a novel, incremental algorithm for projecting a point on to a set of separation constraints.

1 Related Work

The work presented in this paper significantly extends our recent research on handling *band constraints* [3, 4, 5] in stress majorization. As part of this we introduced the DiG-CoLa drawing style for directed graphs in which the nodes are partitioned into an ordered sequence of bands. We also demonstrated the usefulness of band constraints for drawing networks such as rail-networks where the placement of nodes should reflect the underlying geography. Band constraints are a very restricted kind of separation constraints: for instance they cannot model containment, non-overlap or arbitrary vertical separation. Thus, generalizing to separation constraints greatly increases the usefulness of constraint-based stress majorization and opens a number of new applications for constraint-based force-directed layout. In order to handle this greater generality the algorithm we give for projecting on to separation constraints is quite different to that proposed for band constraints [5].

Augmenting force-directed layout with true constraints was first explored by He and Marriott [13, 14], where a Kamada-Kawai-based method was extended with an active-set constraint solving technique to provide separation constraints. However, only small examples of fewer than 20 nodes were tested and the scalability and potential applications of the technique were not examined.

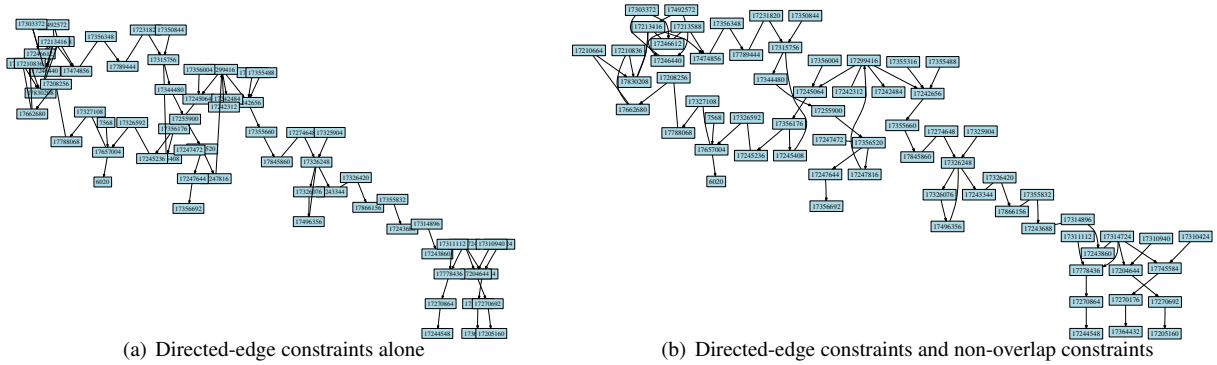
Many researchers have suggested changes to classical force-directed methods to provide some of the functionality of constraints. For example, Ryall et al. [20] added stiff springs to a standard force-directed model to keep user-selected parts of the diagram roughly spaced as desired; various works [12, 22] modified the goal function to discourage node overlaps; while Wang and Miyamoto [22] and Huang and Eades [15] augmenting force-directed models to draw clustered graphs. However, these approaches do not impose strict constraints on the layout, but rather change the force model so as to avoid overlaps or emphasize the partition to clusters etc. Thus there is no guarantee that the constraints are enforced and convergence is often problematic, requiring complex cooling strategies.

2 Applications

2.1 Directed graph layout

Separation constraints can be applied to the problem of arranging a directed graph, where we want to convey hierarchy by orienting all edges ("arrows") in the same direction. If the digraph is acyclic, we can simply define a separation constraint for each edge (u, v) to require that u be positioned above v. If the digraph contains directed cycles we avoid cyclic constraints by omitting constraints for a — preferably minimal — number of edges. Many heuristics are available for solving this *largest acyclic subgraph* problem [9].

This approach results in at most one constraint per edge. This is very different from the band-constraints that we used in the previous DiG-CoLa algorithm [3]. The more restrictive band constraints typically mean that the constrained optimization problem can be solved more quickly, while per-edge constraints can achieve lower *stress* as defined in Eq. (1). As a measure of variation from desired edge length stress gives an indication of layout quality. Figure 1 compares unconstrained layout for a reasonably large graph with layout subject to per-edge separation constraints, DiG-CoLa band constraints and layout by the traditional Sugiyama method [21] for arranging directed graphs. Note that, of the layouts enforcing uniform edge direction DiG-CoLa keeps edge-length relatively consistent but is unable to "unfold" the



(a) Directed-edge constraints alone (b) Directed-edge constraints and non-overlap constraints

Fig. 2. A gene-activation network arranged with directed edge constraints and non-overlap constraints.

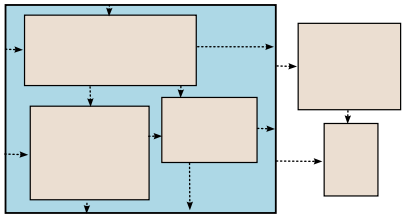


Fig. 3. Illustration of possible separation constraints required to avoid overlap between nodes and to keep nodes within their cluster boundaries. Constraints are shown as arrows.

graph. The Sugiyama strategy is a slight improvement in terms of number of edge crossings but there are a large number of very long edges (in some cases spanning nearly the entire width of the graph) making it difficult to see weakly connected components. Arguably, per-edge separation constraints offer the best available strategy for arranging large and mixed[2] directed graphs, e.g. in Figure 1(b) low stress is coupled with a clear display of the connectivity of the graph and the fewest edge crossings of any of the directed graph drawing methods.

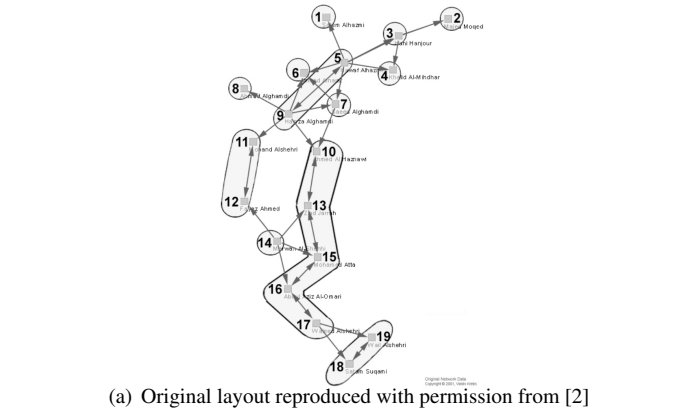
It should be noted that the popular Sugiyama framework is currently significantly faster when applied to large graphs, especially given recent advances [10]. We would argue, however, that run-time scalability is not so useful if the output, when applied to large graphs, gives no insight into the graph structure.

2.2 Graphs with labelled nodes

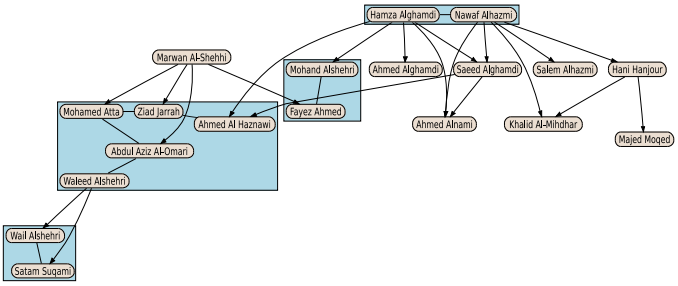
The nodes in graphs modelling real-world relational data usually correspond to some concept or information that needs to be indicated through a textual or graphical label. Separation constraints between the rectangular bounding boxes of labelled nodes give us a way to ensure that these labels do not overlap and hence, remain readable. Previously, overlaps between nodes in graphs arranged by force-directed layout could only be avoided by one of two methods. The first is the so called "layout adjustment" approach which involves a post-processing step that may significantly degrade the quality of the original layout. The second known approach is to add extra repulsive forces between node boundaries. This generally requires complex cooling schedules to achieve a convergent layout and secondly cannot guarantee no overlaps in all cases. By contrast using separation constraints to avoid overlaps avoids modifying the goal function, so that the final layout will still have been optimized subject to the aesthetic criteria of spacing nodes according to the graph path length between them, and guarantees that overlaps can never occur. All the labelled graph examples in this paper are arranged using non-overlap separation constraints.

The layout process is simple. A layout allowing overlaps is first

[2]For graphs with mixed directed and undirected edges no separation constraint needs to be assigned to undirected edges and hence, these can lie horizontally — other layout methods such as Sugiyama force undirected edges to span layers.



(a) Original layout reproduced with permission from [2]

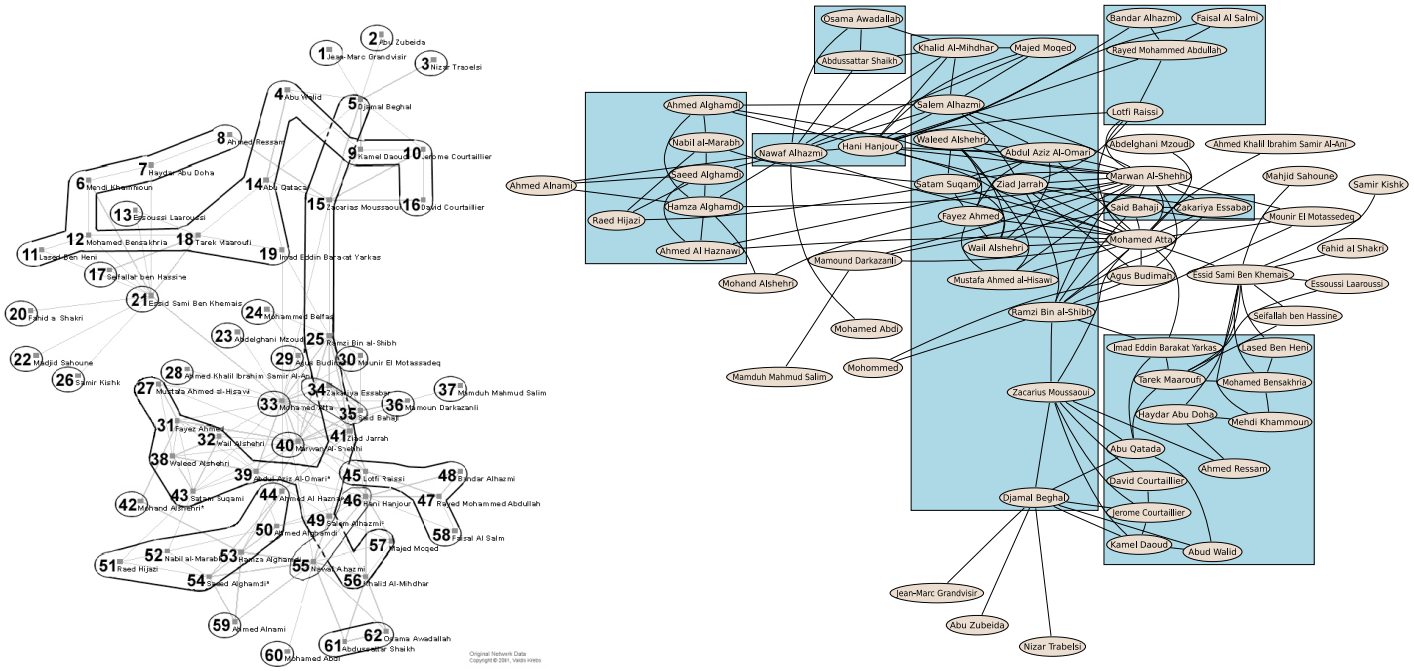


(b) IPSEP-COLA layout using cluster containment constraints and non-overlap constraints

Fig. 4. Mutual influence sets in a terrorist network.

obtained. Stress-majorization layout then continues subject to non-overlap separation constraints generated prior to processing each dimension. Thus, nodes can "slide" past or around each other, but can never overlap. The constraints are generated using a fast scanline algorithm [6] which produces at most $2n$ constraints for a graph with n nodes, e.g. see Figure 3. Note that after removing overlaps the stress-majorization process continues to monotonically decrease stress, meaning that convergence is guaranteed.

Non-overlap constraints can be combined with directed-edge constraints. For example Figure 2 shows a gene activation network where non-overlap constraints and directed edge constraints are combined to produce a compact layout with near uniform edge length, where the precedence of gene activation is clearly shown and individual genes can easily be identified by their labels. A version without non-overlap constraints is also shown for contrast. Note that the gap required by the non-overlap constraints is set slightly larger than the actual node size. This leaves some space through which edges can be routed.



(a) Conventional force-directed layout (reproduced with the authors' permission from [2])

(b) IPSEP-COLA layout using cluster containment constraints and non-overlap constraints

Fig. 5. Mutual influence sets in a larger terrorist network.

2.3 Clustered graphs

Separation constraints can also be used to require *containment* within some boundary. One possible application of such constraints is to require layout within the fixed width and height bounds of a page or arbitrarily sized window. Another example is layout where arbitrary groups of nodes — or *clusters* — are required to be placed within certain bounds. Combining these containment constraints with non-overlap constraints (as illustrated in Figure 3) we are able to produce high-quality drawings of clustered graphs. Specifying constraints between nodes and cluster boundaries requires the use of dummy variables for each of left, right, top and bottom positions. We have also found that it is useful to specify a small attraction between the left and right, top and bottom pairs of dummy variables by trivial modification of the A matrix of Eq. (2). This helps to keep clusters compact relative to the rest of the graph.

Figures 4 and 5 show a possible application of this type of clustered graph layout. The data is a terrorist communication network studied in detail in Brams et al. [2]. The figures appearing in the original paper were arranged using unconstrained force-directed layout. The authors identified subgraphs within which two-way communication (mutual influence) was observed between the terrorist suspects. Two of the original figures of Brams et al. are shown in Figures 4(a) and 5(a). Note that the mutual influence subgraphs, highlighted presumably manually, fall within non-convex boundaries that can overlap other parts of the graph. We would argue that in our rendering of the same networks (Figures 4(b) and 5(b)) the mutual influence cells are more easily identifiable. Notice also that in Figure 4(b) separation constraints have been used to force directed edges to point downwards while within the mutual influence networks, where the edges are bidirectional, no such constraints are applied. This helps to show the hierarchy within the terrorist organization.

2.4 Directed and Clustered Multi-dimensional Scaling

Previously we introduced *Directed Multidimensional Scaling* (DMDS) as a way of displaying an ordered classification of high-dimensional data over a multidimensional-scaling plot [3]. Figure 6 shows a DMDS plot for breakfast cereal data, with separation constraints used to enforce a total order—based on a dietician's health rating—over the cereals. Note that for n data points $n-1$ constraints are required to enforce a total ordering. Also new in this figure is the use of separation constraints to prevent overlapping labels. In our original figure [3] overlaps had to be removed by hand to make the labels readable.

The non-overlap and containment constraints used above to draw clustered graphs give us yet another possibility for enriching an MDS data plot. In Figure 7 we use such constraints to group the cereals by manufacturer in addition to the directional constraints highlighting health rating. This Clustered MDS (CMDS) plot allows us to easily see if manufacturers specialize in certain types of products. For example we might conclude that Nabisco generally produces healthier cereals than Post while Kelloggs appears to produce a wide variety of cereals.

Clearly the more constraints that are imposed on an MDS plot the more difficult it is to achieve a low stress value. The final stress value for a DMDS or CMDS plot can be used as a quality measure for the *fit* of the constraints to the data. Thus, we suggest that constraints can be imposed on an MDS plot to test a theory about the structure of the data and the final stress value could be considered a measure of the accuracy of the theory.

3 Algorithm

3.1 Problem formulation

The general goal or *stress* function that we seek to minimize is:

$$stress(X) = \sum_{i<j} w_{ij}(||X_i - X_j|| - d_{ij})^2 \qquad (1)$$

where for each pair of nodes i and j, d_{ij} gives an ideal separation between i and j (usually their graph-theoretical distance), $w_{ij} = d_{ij}^{-2}$ is used as a normalization constant and X is a $n \times d$ matrix of positions for all nodes, where d is the dimensionality of the drawing and n is the number of nodes.

Majorization minimizes this stress function by iteratively minimizing quadratic forms that approximate and bound it from above.

At each iteration we determine the coordinates x of the nodes in each dimension by minimizing $f(x) = \frac{1}{2}x^T Ax - x^T b$ where there are n

Fig. 6. Various cereal brands arranged using Directed Multi-Dimensional Scaling, i.e. separation of the cereals is based on the similarity of their nutritional data while directed edge constraints have been used to require that the healthier cereals (as determined by a dietitian's ranking) must be higher than less healthy cereals. Red coloring indicates high sugar content.

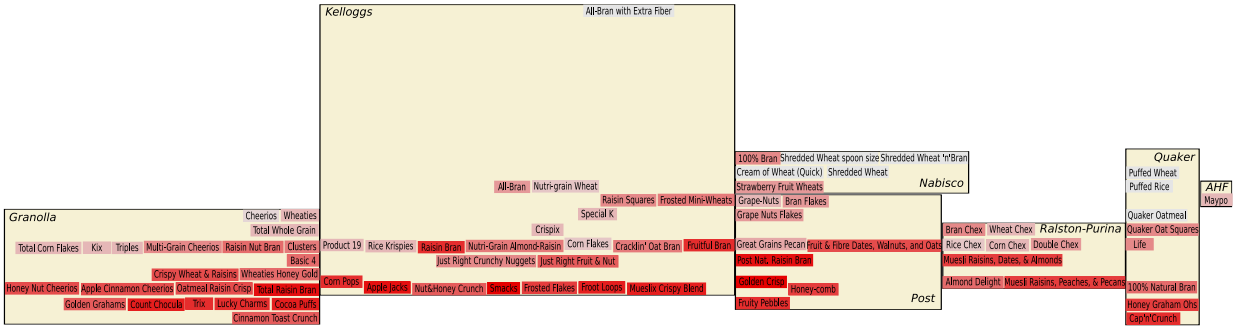


Fig. 7. Various cereal brands arranged using Clustered Multi-Dimensional Scaling, clustered by manufacturer.

nodes $A \in \mathbb{R}^n \times \mathbb{R}^n$ is positive semi-definite matrix and $x, b \in \mathbb{R}^n$. The vector x is a column of X such that x_i is the coordinate of node i. The reader is referred to [11] for the definition of A and b.

In this paper we consider the case where we have additional separation constraints. A *separation constraint* c is of form $u + a \leq v$ where u, v are variables and a is the minimum gap between them. We use the notation $left(c)$, $right(c)$ and $gap(c)$ to refer to u, v and a respectively. We require that constraints are axis separable in the sense that they always constrain variables in the same dimension.

We can treat a set of separation constraints C over variables V as a weighted directed graph with a node for each $v \in V$ and an edge for each $c \in C$ from $left(c)$ to $right(c)$ with weight $gap(c)$. We call this the *constraint graph*. We define $out(v) = \{c \in C \mid left(c) = v\}$ and $in(v) = \{c \in C \mid right(c) = v\}$. Note that edges in this graph are *not* the edges in the original graph.

We restrict attention to problems in which the constraint graph is acyclic, and for which there is at most one edge between any pair of variables. It is possible to transform any satisfiable set of separation constraints into an equivalent problem of this form as long as the gaps are non-negative. Thus, at each iteration step and for each dimension in the drawing we solve:

$$\min_x \frac{1}{2} x^T A x - x^T b \text{ subject to: } C \tag{2}$$

We call this the Quadratic Programming with Separation Constraints (QPSC) problem.

Since A is positive semi-definite, the problem has only global minima. Such a QP problem can be solved in polynomial time [19]. However, our experiments detailed in Section 3.3 show that generic QP solvers are quite slow at solving this problem. To accelerate computation we utilize two special characteristics of the problem:

1. During the majorization process, we iteratively solve closely related QPs: The matrix A is not changed between iterations, only the vector b is changed, while the constraints either remain unchanged or are slightly modified. We ensure that changes to the constraints maintain feasibility of the current solution. Therefore, the solution of the previous iteration is still a feasible solution for current iteration. Moreover, this previous solution is probably very close to the new optimal solution (e.g., consider that in most iterations the coordinates are only slightly changed). However, such "warm-start" initialization is fundamentally not trivial for the barrier (or interior-point) methods used by most commercial solvers.
2. Standard QP solvers allow general linear constraints. Separation constraints are very simple as each of them involve only two variables. Importantly, we can develop a method for solving (2) which takes advantage of this restricted form of constraint. Consequently, in the next section we describe a specialized algorithm for solving the QPSC problem.

3.2 Gradient Projection Algorithm

We give an iterative *gradient-projection* algorithm [19, pp. 476–481] for finding a solution to a QPSC Problem. The algorithm, *solve_QPSC*, is shown in Figure 8. It first decreases $f(x)$, by moving

x in the direction of steepest descent, i.e. the opposite of the gradient $\nabla f(x) = Ax + b$. For the moment ignore the call to the procedure *split_blocks*. While we are guaranteed that — with appropriate selection of step-size s — the energy is decreased by this first step, the new positions may violate the constraints. We correct this by calling *project*, which returns the closest point $\bar{x}$ to x which satisfies the separation constraints, i.e. it projects x on to the feasible region. Finally, we calculate a vector d from our initial position $\hat{x}$ to $\bar{x}$ and we ensure monotonic decrease in stress when moving in this direction by computing a second stepsize $\alpha = \arg\min_{\alpha \in [0,1]} f(x + \alpha d)$ which minimizes stress in this interval.

procedure *solve_QPSC(A, b, C)*
  repeat
    $g \leftarrow Ax + b$
    $s \leftarrow \frac{g^T g}{g^T A g}$
    $\hat{x} \leftarrow x$
    $x \leftarrow \hat{x} - sg$
    $nosplit \leftarrow split\_blocks(x)$
    $\bar{x} \leftarrow project(C)$
    $d \leftarrow \bar{x} - \hat{x}$
    $\alpha \leftarrow \max(\frac{g^T d}{d^T A d}, 1)$
    $x \leftarrow \hat{x} + \alpha d$
  until $\|\hat{x} - x\|$ sufficiently small and *nosplit*
  return x

Fig. 8. Algorithm to find an optimal solution to a QPSC problem with variables $x_1, \ldots, x_n$, symmetric positive-semidefinite matrix A, vector b and separation constraints C over the variables.

While the algorithm given in Figure 8 describes a fairly standard gradient-projection approach, the procedures *project* and *split_blocks* are the part of the algorithm specific to our particular QP. The main difficulty in implementing gradient-projection methods is the need to efficiently project on to the feasible region.

The projection operation essentially requires solving a QP of the form $\min_x \sum_{i=1}^n (x_i - p_i)^2$ subject to the separation constraints C where $p = \hat{x} - sg$ is defined in the gradient-projection step. The *project* procedure (Figure 9) iteratively changes the variable's positions till all constraints are satisfied. The algorithm works by merging variables into larger and larger "blocks" of contiguous variables connected by a spanning tree of active constraints, where a separation constraint $u + a \leq v$ is active if for the current position for u and v, $u + a = v$.

We represent a block B_i using a record with the following fields: *vars*, the set of variables in the block; *nvars*, the number of variables in the block; *active*, the set of constraints between variables in the block which form the spanning tree of active constraints; *posn*, the position of the block's "reference point".

The algorithm also uses two arrays *blocks* and *offset* indexed by variables where $block_i$ gives the block of variable i and $offset_i$ gives the distance from i to its block's reference point. We have that the current position of variable i, $posn(i)$, is given by the expression $B_{block_i}.posn + offset_i$.

On each call to *solve_QPSC* we start from the blocks computed in the previous call to *solve_QPSC* for that dimension and from the previously computed value for x. At the very start of the process we put each variable i in its own block and place it at x_i.

The *project* procedure starts with the current set of blocks and moves them to their new optimal position. An invariant of the algorithm is that the reference point $B.posn$ for each block B is at its optimal position which is simply the average of the variables in the block's desired positions appropriately translated to the same reference frame:

$$\frac{\sum_{i \in B.vars} x_i - offset_i}{B.nvars}.$$

Of course moving the blocks may mean that some constraints are now violated. We repeatedly find the most violated constraint c where

procedure *project(C)*
$c \leftarrow \max_{c \in C} violation(c)$
while $violation(c) \geq 0$ **do**
  if $block_{left(c)} \neq block_{right(c)}$ **then**
    merge_block($block_{left(c)}$, $block_{right(c)}$, c)
  else expand_block($block_{left(c)}$,c)
  $c \leftarrow \max_{c \in C} violation(c)$
for $i = 1, \ldots, n$ **do**
  $x_i \leftarrow B_{block_i}.posn + offset_i$
return x

procedure *merge_block(L, R, c)*
$d \leftarrow offset_{left(c)} + gap(c) - offset_{right(c)}$
$B_L.posn \leftarrow \frac{B_L.posn \times B_L.nvars + (B_R.posn - d) \times B_R.nvars}{B_L.nvars + B_R.nvars}$
$B_L.active \leftarrow B_L.active \cup B_R.active \cup \{c\}$
for $i \in B_R.vars$ **do**
  $block_i \leftarrow L$
  $offset_i \leftarrow d + offset_i$
$B_L.vars \leftarrow B_L.vars \cup B_R.vars$
$B_L.nvars \leftarrow B_L.nvars + B_R.nvars$
$B_R.nvars \leftarrow 0$
return

procedure *expand_block*$(b, \tilde{c})$
for each $c \in B_b.active$ **do** $lm_c \leftarrow 0$ **end for**
$AC \leftarrow B_b.active$
$comp\_dfdv(left(\tilde{c}), AC, NULL)$
$[v_1, \ldots, v_k]$:= $comp\_path(left(\tilde{c}), right(\tilde{c}), AC)$
$ps \leftarrow \{c \in AC \mid \exists j \text{ s.t. } left(c) = v_j \text{ and } right(c) = v_{j+1}\}$
$sc \leftarrow \min_{c \in ps} lm_c$
$AC \leftarrow AC \setminus \{sc\}$
for each $v \in connected(right(\tilde{c}), AC)$ **do**
  $offset_v \leftarrow offset_v + violation(\tilde{c})$
$B_b.active \leftarrow AC \cup \{\tilde{c}\}$
$B_b.posn \leftarrow \frac{\sum_{j \in B_b.vars} x_j - offset_i}{B_b.nvars}$
return

function *comp_dfdv*(v, AC, u)
$dfdv \leftarrow posn(v) - x_v$
for each $c \in AC$ s.t. $v = left(c)$ and $u \neq right(c)$ **do**
  $lm_c \leftarrow comp\_dfdv(right(c), AC, v)$
  $dfdv \leftarrow dfdv + lm_c$
for each $c \in AC$ s.t. $v = right(c)$ and $u \neq left(c)$ **do**
  $lm_c \leftarrow -\ comp\_dfdv(left(c), AC, v)$
  $dfdv \leftarrow dfdv - lm_c$
return $dfdv$

procedure *split_blocks*(x)
$nosplit \leftarrow true$
for each block B_i s.t. $B_i.nvars > 0$ **do**
  $B_i.posn \leftarrow \frac{\sum_{j \in B_i.vars} x_j - offset_j}{B_i.nvars}$
  $AC \leftarrow B_i.active$
  for each $c \in AC$ **do** lm_c := 0 **end for**
  choose $v \in B_i.vars$
  $comp\_dfdv(v, AC, NULL)$
  $sc \leftarrow \min_{c \in AC} lm_c$
  if $lm_c \geq 0$ **then break**
  $nosplit \leftarrow false$
  $AC \leftarrow AC \setminus \{sc\}$
  $s \leftarrow right(sc)$
  $B_s.vars \leftarrow connected(s, AC)$
  for each $v \in B_s.vars$ **do** $block_v$:= s **end for**
  $B_i.vars \leftarrow B_i.vars \setminus B_s.vars$
  $B_s.nvars \leftarrow |B_s.vars|$, $B_i.nvars \leftarrow |B_i.vars|$
  $B_s.posn \leftarrow \frac{\sum_{j \in B_s.vars} x_j - offset_j}{B_s.nvars}$, $B_i.posn \leftarrow \frac{\sum_{j \in B_i.vars} x_j - offset_i}{B_i.nvars}$
  $B_i.active \leftarrow \{c \in AC \mid left(c) \in B_s.vars \text{ and } right(c) \in B_s.vars\}$
  $B_s.active \leftarrow AC \setminus B_i.active$
return $nosplit$

Fig. 9. Algorithm to project variables to the closest feasible position.

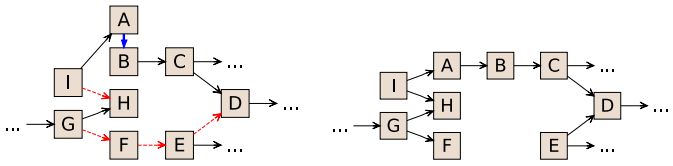


Fig. 10. An example of the use of *expand_block*. Here, we are drawing a directed graph horizontally with the separation constraint that $u+1 \leq v$ if there is an edge from u to v. Initially, the nodes A,B,C,D,E,F,G,H,I form a block connected by the active tree of constraints shown as black and red arrows. Now the edge from A to B is discovered to be violated. This is shown in blue in the top diagram. Since A and B are in the same block *expand_block* is called. The core function of *expand_block* is to determine where to split the current active tree of constraints to allow the edge (A,B) to be inserted. Edges which are valid split points shown in red dashes. Assuming that the valid split point with the smallest Lagrange multiplier is the edge (F,E), this edge will be removed from the active constraints being replaced by (A,B) to give the layout shown in the bottom.

$violation(c) = posn(left(c)) + gap(c) - posn(right(c))$. If c connects two different blocks B_L and B_R then we merge the two blocks connected by c using the function $merge\_block(L,R,c)$. This function adds block B_R to B_L with c as the active connecting constraint. It appropriately computes the new position for the block in terms of the positions of B_R and B_L. If c connects two variables in the same block B_b, then we use $expand\_block(b,c)$ which pushes the variables in the block apart by making c active. We repeat this until no constraint is violated in which case we have (almost) projected on to the constraints.

The procedure $expand\_block(b,\tilde{c})$ is the most complex part of the algorithm. An example of its use is shown in Figure 10. It deals with a case where a previously constructed block now causes a constraint $\tilde{c}$ between two variables in the block to be violated. To fix this we must identify where to split the current block and then rejoin the sub-blocks using $\tilde{c}$. This "expands" the block to remove the violation, by spanning it using a different spanning tree of active constraints. More, precisely, we first compute the best constraint sc in the active set on which to split. To do this we compute the Lagrange multiplier lm_c for each active constraint c in the block using the procedure *comp_dfdv* introduced in [6]. Lagrange multipliers are a fundamental notion in constrained optimization, but describing their properties is beyond the scope of this paper; the interested reader is referred to [19]. Here, it suffices to understand that the value of lm_c gives the rate of increase of the goal function as a function of $right(c) - left(c)$. Thus the smaller the value of lm_c the better it is to split the block at that constraint. However, not all constraints in the active set are valid points for splitting. Clearly we must choose a constraint that is on the path between the variables $left(\tilde{c})$ and $right(\tilde{c})$. The call to function *comp_path* returns the list of variables $[v_1,...,v_k]$ on the path from $left(\tilde{c})$ to $right(\tilde{c})$ along the constraints in the active set of constraints AC. Furthermore, to be a valid split point the constraint c must be oriented in the same direction as $\tilde{c}$, i.e. for some j, $left(c) = v_j$ and $right(c) = v_{j+1}$. The split constraint sc is simply the valid split constraint with the least Lagrange multiplier. The remainder of *expand_block* splits the block by removing sc from the active set AC and then moves each variable in the right block, i.e. those connected to $right(\tilde{c})$, to the right by the amount required to fix the violation of $\tilde{c}$. The constraint $\tilde{c}$ is added to the active set for the block to rejoin the two sub-blocks and the block is placed at its optimal location.

As we have alluded, the call $project(C)$ is not guaranteed to perform an exact projection of x onto C. It will always return a solution satisfying C but this may not be the closest feasible point to x. The problem is that $project(C)$ can perform a merge which is later made unnecessary, but *project* never splits blocks even if this leads to a better solution.

In practice it is relatively rare for this to happen. We fix this problem lazily using the procedure *split_blocks*. This takes the blocks from the previous iteration and places them at their new position. It then identifies which previous merges need to be undone by computing the Lagrange multiplier lm_c for all active constraints c. If all of these are non-negative then no block needs to be split. Otherwise, for each block B_i with a negative Lagrange multiplier we choose the constraint sc with the most negative lm_c and remove this from the active set AC for the block. The variables connected to $right(sc)$ are removed from B_i and placed in a new block $B_{right(sc)}$. The other fields are appropriately updated. This ensures that on termination a locally optimal solution is found.

We have previously defined the problem of adjusting a graph layout so that all overlaps between nodes are removed as the solution to a QP with separation constraints [6]. In fact that QP was the same type of sum-of-least-squares problem that is required in the projection step of a constrained majorization algorithm involving separation constraints. That is, the problem involved finding a solution that minimized the sum of squared displacements of all nodes — in the same way that in the projection step of gradient projection we need to displace each variable by as little as possible in order to satisfy the constraints. However our previous algorithm was non-incremental, slow if an exact projection was required, and complicated to implement (a number of special cases must be handled for correct behavior [7]). The algorithm given above is quite different, incremental, reasonably fast and asymptotically exact.

It is worth pointing out that the worst case complexity of *solve_QPSC* is $O((n+m)^2)$ where n is the number of variables and m the number of constraints. Typically the number of constraints is linear in the number of variables in which case the algorithm is quadratic in the number of variables. This is the same as the complexity in the unconstrained case and the complexity of our dedicated solver used for solving band constraints [5].

The complexity is computed as follows. Since A is an $n \times n$ matrix, the complexity of computing g, s and α is $O(n^2)$. The complexity of *split_blocks* is $O(n)$ since the total number of active constraints is $O(n)$ as they form a spanning forest for the variables. The complexity of *project* is $O(m \times (n+m)+n)$. The main **while** loop can only be called $O(m)$ times since once a constraint is made active it can never become violated again. This is because if it is active then it cannot be violated because of the way blocks are constructed. It can only be made inactive by a subsequent call to *expand_block* in which case it will be implied by the other active constraints and so cannot become violated since these cannot be violated. At each iteration of the main **while** loop the most violated constraint must be determined. Currently we scan through all non-active constraints which has cost $O(m)$. We plan to investigate more efficient data structures in the future. Both *merge_block* and *expand_block* are $O(n)$ in the number of variables in the blocks being merged or expanded. Clearly the last for loop has $O(n)$ complexity.

3.3 Algorithm Performance

To test the efficiency of the *solve_QPSC* algorithm we used it to arrange a set of graphs using directed edge constraints. Our test data is a set of 52 graphs from the Matrix Market [1], the AT&T Graphs collection[3] and other sources[4]. The graphs were chosen to be representative of different application domains (software design diagrams, gene activation networks, citation networks, etc.) and different types of graph structure including both dense and sparse graphs, regular meshes and scale-free networks.

We compared running time for constrained layout using the *solve_QPSC* algorithm and the commercial Mosek quadratic program solver [18]. We also compare against unconstrained layout using the conventional conjugate gradient method to solve the quadratic subproblems. The results are shown graphically in Figure 11. The independent variable is edge count for each sample graph, which corresponds to the number of constraints required and should therefore vary proportionally to running time and stress.

From the chart it is easy to see that the purpose designed *solve_QPSC* solver is faster than Mosek in all cases, usually by at least

[3] `ftp://ftp.research.att.com/dist/drawdag/dg.gz`

[4] Our test data is available from `www.csse.monash.edu.au/~tdwyer`

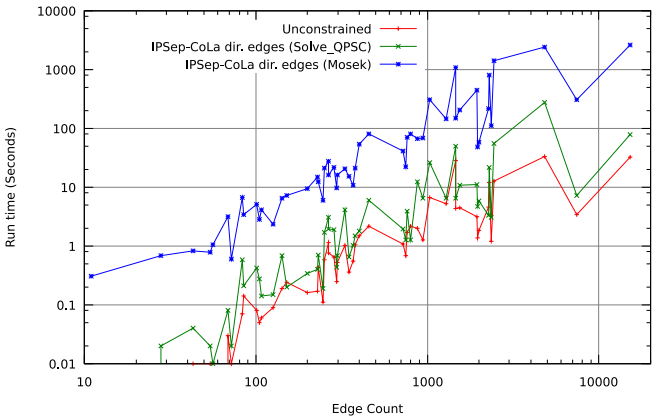


Fig. 11. Running time for unconstrained layout and layout with directed edge constraints using either *solve_QPSC* or the Mosek solver to solve the quadratic program.

a factor of 10 (note the log scale). Generally *solve_QPSC* is somewhat slower than unconstrained layout. The exceptions occur when a constrained local minimum is reached in relatively few iterations. The trade-off is that the constrained solution has higher stress and may therefore be less readable.

4 Discussion

An obvious limitation of the IPSEP-COLA method is that separation constraints must be linear and orthogonal to the axes. This is a problem when nodes have non-rectangular boundaries. Also, clustered graphs could be arranged more compactly if clusters could be fitted with non-overlapping convex hulls rather than simple rectangles. It is possible—though somewhat complicated and computationally expensive—to approximate arbitrary linear constraints with separation constraints by modifying the goal function slightly. Non-linear constraints can be approximated by a sequence of line-segments. We have recently explored such a process with application to edge routing [8].

Another limitation, as described in Section 2.1, is that constraints must be satisfiable, for example, care may need to be taken to avoid generating cyclic constraints. We are working on a modified solver that can detect cycles and relax the constraints involved until they can be satisfied. Also, the more constraints that are placed on layout the greater the problem of local minima. Generally, for highly constrained graphs we would suggest a preprocessing step such as unconstrained layout or a method for quickly finding a feasible arrangement.

The greatest advantage of IPSEP-COLA over other graph drawing algorithms is its flexibility. The mapping of drawing conventions to constraints is potentially simple enough that users of network visualization and diagramming tools can define constraints interactively. Such interactivity is further facilitated by the incremental nature of the method. That is, the constrained stress majorization process can begin from any input arrangement and the layout process animated to preserve the mental map. Finally, IPSEP-COLA provides a single generic technique for arranging real-world networks using the most common drawing conventions. This contrasts strongly with the current state of affairs in which different drawing conventions and layout constraints require very different algorithms.

5 Acknowledgements

We have implemented IPSEP-COLA in the Neato layout tool distributed with the Graphviz package from AT&T. Thanks to Emden Gansner for incorporating our changes into the Graphviz code repository. We wish to thank Stephen J. Brams for permission to use the terrorist networks shown in Figures 4 and 5.

References

[1] R. Boisvert, R. Pozo, K. Remington, R. Barrett and J. Dongarra, "The Matrix Market: A web resource for test matrix collections", in *Quality of Numerical Software, Assessment and Enhancement*, Chapman Hall (1997), pp. 125–137.

[2] S. J. Brams, H. Mutlu and S. L. Ramirez, "Influence in Terrorist Networks: From Undirected To Directed Graphs", *Studies in Conflict and Terrorism* **29(7)**, Taylor and Francis (to appear 2006).

[3] T. Dwyer and Y. Koren, "DIG-COLA: Directed Graph Layout through Constrained Energy Minimization", *IEEE Symposium on Information Visualization (Infovis'05)*, (2005), pp. 65–72.

[4] T. Dwyer, Y. Koren and K. Marriott, "Drawing Directed Graphs Using Quadratic Programming", *IEEE Transactions on Visualization and Computer Graphics* **12(4)**, IEEE (2006), pp. 536–548.

[5] T. Dwyer, Y. Koren and K. Marriott, "Stress Majorization with Orthogonal Ordering Constraints", *Proc. 13th Int. Symp. Graph Drawing (GD'05)*, LNCS **3843**, Springer (2006), pp. 141–152.

[6] T. Dwyer, K. Marriott and P. J. Stuckey, "Fast Node Overlap Removal", *Proc. 13th Int. Symp. Graph Drawing (GD'05)*, LNCS **3842**, Springer (2006), pp. 153–164.

[7] T. Dwyer, K. Marriott and P. J. Stuckey, "Fast Node Overlap Removal — Addendum", *Technical Report*, Monash University, http://www.csse.monash.edu.au/∼tdwyer/FNORAddendum.pdf.

[8] T. Dwyer, K. Marriott and M. Wybrow, "Integrating Edge Routing into Force-Directed Layout", *Proc. 14th Int. Symp. Graph Drawing (GD'06)*, Springer (to appear 2007).

[9] P. Eades and X. Lin, "A New Heuristic for the Feedback Arc Set Problem", Australian Journal of Combinatorics **12**, (1995), pp. 15–26.

[10] M. Eiglsperger, M. Siebenhaller and M. Kaufmann, "An Efficient Implementation of Sugiyama's Algorithm for Layered Graph Drawing", Journal of Graph Algorithms and Applications **9(5)** (2005), pp. 305–325.

[11] E. Gansner, Y. Koren and S. North, "Graph Drawing by Stress Majorization", *Proc. 12th Int. Symp. Graph Drawing (GD'04)*, LNCS 3383, Springer (2004), pp. 239–250.

[12] D. Harel and Y. Koren, "Drawing Graphs with Non-uniform Vertices", *Proc. Working Conference on Advanced Visual Interfaces (AVI'02)*, ACM Press (2002), pp. 157-166.

[13] W. He and K. Marriott, "Constrained Graph Layout", *Proc. 3rd Int. Symp. Graph Drawing (GD'96)*, LNCS **1190**, Springer (1996), pp. 217–232.

[14] W. He and K. Marriott, "Constrained Graph Layout", *Constraints* **3** (1998), pp. 289–314.

[15] M.L. Huang and P. Eades, "A Fully Animated Interactive System for Clustering and Navigating Huge Graphs", *Proc. 5rd Int. Symp. Graph Drawing (GD'98)*, LNCS **1547**, Springer (1998), pp. 374–383

[16] T. Kamada and S. Kawai, "An Algorithm for Drawing General Undirected Graphs", *Information Processing Letters* **31** (1989), pp. 7–15.

[17] K. Misue, P. Eades, W. Lai, and K. Sugiyama, "Layout Adjustment and the Mental Map", *Journal of Visual Languages and Computing* **6** (1995), pp. 183–210.

[18] Mosek Optimization Toolkit V3.2 `www.mosek.com`.

[19] J. Nocedal and S. Wright, *Numerical Optimization*, Springer (1999).

[20] K. Ryall, J. Marks and S. M. Shieber, "An Interactive Constraint-Based System for Drawing Graphs", *ACM Symposium on User Interface Software and Technology* (1997), pp. 97–104.

[21] K. Sugiyama, S. Tagawa and M. Toda, "Methods for Visual Understanding of Hierarchical Systems", *IEEE Trans. Systems, Man, and Cybernetics* **11** (1981), pp. 109–125.

[22] X. Wang, and I. Miyamoto, "Generating Customized Layouts", *Proc. 2nd Int. Symp. Graph Drawing (GD'95)*, LNCS **1027**, pp. 504–515.