# Axis-by-Axis Stress Minimization

Yehuda Koren and David Harel

Dept. of Computer Science and Applied Mathematics
The Weizmann Institute of Science, Rehovot, Israel
{yehuda,dharel}@wisdom.weizmann.ac.il

**Abstract.** Graph drawing algorithms based on minimizing the so-called stress energy strive to place nodes in accordance with target distances. They were first introduced to the graph drawing field by Kamada and Kawai [11], and they had previously been used to visualize general kinds of data by multidimensional scaling. In this paper we suggest a novel algorithm for the minimization of the Stress energy. Unlike prior stress-minimization algorithms, our algorithm is suitable for a one-dimensional layout, where one axis of the drawing is already given and an additional axis needs to be computed. This 1-D drawing capability of the algorithm is a consequence of replacing the traditional node-by-node optimization with a more global axis-by-axis optimization. Moreover, our algorithm can be used for multidimensional graph drawing, where it has time and space complexity advantages compared with other stress minimization algorithms.

## 1 Introduction

A graph is a structure $G(V=\{1,\ldots,n\},E)$ representing a binary relation $E$ over a set of nodes $V$. In [9] we addressed the problem of drawing a graph in one dimension. The major application is when one of the drawing coordinates is already given, and we want to compute an additional coordinate for each node. Traditional force-directed algorithms turned out to be unsuitable in this 1-D case, because of their local optimization methods, which become stuck in bad local minima. In this paper, we show how the familiar method of Kamada and Kawai [11], which minimizes an energy function known as *the stress energy*,[1] can be used for 1-D graph drawing. We introduce a novel method for axis-by-axis optimization of the stress energy, which is very suitable for 1-D graph drawing as it utilizes the special nature of this problem. Interestingly, using our technique it seems that minimizing the stress in only one dimension is definitely the easier case for stress minimization. Moreover, we harness the benefits of the new 1-D-based algorithm for computing a multidimensional drawing of the graph, by calculating each axis of the drawing separately. In this multidimensional case, our algorithm also has an advantage over the classical stress minimization suggested by Kamada and Kawai.

## 2 One-Dimensional Stress Optimization

*Stress energy* is a traditional measure of drawing quality, based on the heuristic that a nice drawing relates to good isometry. Accordingly, it calls for placing the nodes so that the resulting pairwise Euclidean distances will approach the corresponding target

[1] Indeed, the stress energy was originally used for multidimensional scaling, where its name originated; see, e.g., [3, 12].

(graph-theoretical) distances. Given a $k$-D layout $x = (x_1, \ldots, x_n)$, where the place of node $i$ is $x_i \in \mathbb{R}^k$, the concrete form of the energy is:

$$E(x) \stackrel{\text{def}}{=} \sum_{i<j} k_{ij} \left(|x_i - x_j| - d_{ij}\right)^2 . \tag{1}$$

Here, the target distance $d_{ij}$ is typically the graph-theoretical distance between nodes $i$ and $j$. The normalization constant $k_{ij}$ equals $d_{ij}^{-\alpha}$, where $0 \leqslant \alpha \leqslant 2$. Kamada and Kawai [11] picked $\alpha = 2$, whereas Cohen [3] also considered $\alpha = 0$ and $\alpha = 1$. Moreover, Cohen suggested setting $d_{ij}$ to the linear-network distance.

For the rest of this paper, we explore axis-by-axis stress minimization. Hence, we assume $x$ to be an 1-D layout, i.e., $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$. Given two 1-D layouts, $x, \tilde{x} \in \mathbb{R}^n$, we define the following family of auxiliary functions:

$$\delta_{ij}^{\tilde{x}}(x) = \begin{cases} x_i - x_j & \tilde{x}_i \geqslant \tilde{x}_j \\ x_j - x_i & \tilde{x}_i < \tilde{x}_j \end{cases} \qquad 1 \leqslant i < j \leqslant n. \tag{2}$$

Next, we define the following energy function of the layout $x$:

$$E^{\tilde{x}}(x) \stackrel{\text{def}}{=} \sum_{i<j} k_{ij} \left(\delta_{ij}^{\tilde{x}}(x) - d_{ij}\right)^2 . \tag{3}$$

It is important to understand the relations between $E^{\tilde{x}}(x)$ and the stress energy, $E(x)$.

**Lemma 1.** *For every $x, \tilde{x} \in \mathbb{R}^n$, $E(x) \leqslant E^{\tilde{x}}(x)$.*

*Proof.* Let us pick some pair $i < j$, and analyze the corresponding terms of $E(x)$ and $E^{\tilde{x}}(x)$. Observe that $|\delta_{ij}^{\tilde{x}}(x)| = |x_i - x_j|$. In addition, it is always the case that $k_{ij}, d_{ij} \geqslant 0$. Therefore, $k_{ij}(\delta_{ij}^{\tilde{x}}(x) - d_{ij})^2 \geqslant k_{ij}(|x_i - x_j| - d_{ij})^2$. The lemma follows. □

**Lemma 2.** *For every $x \in \mathbb{R}^n$, $E(x) = E^x(x)$.*

*Proof.* Simply observe that $\delta_{ij}^{x}(x) = |x_i - x_j|$. □

By the last two Lemmas we conclude:

**Corollary 1.** *For every $x, \tilde{x} \in \mathbb{R}^n$, $E^x(x) \leqslant E^{\tilde{x}}(x)$.*

The usefulness of the energy $E^{\tilde{x}}(x)$ stems from the fact that it can be minimized optimally. To realize this we need some additional notations. First, we define a related $n \times n$ Laplacian matrix $\mathcal{L}$, where

$$\mathcal{L}_{ij} = \begin{cases} -k_{ij} & i \neq j \\ \sum_{j \neq i} k_{ij} & i = j \end{cases} \qquad i, j = 1, \ldots, n.$$

Note that the Laplacian depends only on the graph, regardless of its layout $\tilde{x}$. We also use the vector $b^{\tilde{x}} \in \mathbb{R}^n$, where:

$$b_i^{\tilde{x}} = \sum_{j \neq i:\ \tilde{x}_j \leqslant \tilde{x}_i} k_{ij} d_{ij} - \sum_{j \neq i:\ \tilde{x}_j > \tilde{x}_i} k_{ij} d_{ij} \qquad i = 1, \ldots, n.$$

Now, using some elementary algebra, it can be shown that:

$$E^{\tilde{x}}(x) = x^T \mathcal{L} x - 2x^T b^{\tilde{x}} + C\,,$$

where $C$ is a constant that is independent of $x$. Since the Laplacian is known to be positive semi-definite, we conclude by differentiation:

**Lemma 3.** *The minimizer of $E^{\tilde{x}}(x)$ is the solution of the system of equations:*

$$\mathcal{L}x = b^{\tilde{x}}\,.$$

All these observations suggest the following process for minimizing the stress energy:

**Function 1-D_stress_minimization** ($G(V,E),\ x \in \mathbb{R}^n$)
  Compute the Laplacian $\mathcal{L}$
  **do**
    $\tilde{x} \leftarrow x$
    Compute $b^{\tilde{x}}$
    Compute $x$ for which $\mathcal{L}x = b^{\tilde{x}}$
  **while** ($x \neq \tilde{x}$)

Our main result is that each iteration of this process (except for the last one) must decrease the stress energy: $E(x) < E(\tilde{x})$. This stems from the above lemmas: By Lemma 2, $E(\tilde{x}) = E^{\tilde{x}}(\tilde{x})$, and by Lemma 3, $E^{\tilde{x}}(x) < E^{\tilde{x}}(\tilde{x})$ (since $\tilde{x} \neq x$, the energy must decrease). Now, by Corollary 1, $E^x(x) \leqslant E^{\tilde{x}}(x)$. Taken together, we obtain:

$$E(x) = E^x(x) \leqslant E^{\tilde{x}}(x) < E^{\tilde{x}}(\tilde{x}) = E(\tilde{x})\,.$$

Of course the energy is bounded below by zero, so the process must converge. Unlike node-by-node local optimization methods, the new optimization process does not suffer from working in a single dimension. Later, we introduce some encouraging experimental results. However, note that like other optimization methods, we can only guarantee convergence to a local minimum.

### 2.1 Working with a sparse Laplacian

A significant drawback of stress optimization is that its space complexity is $\Theta(n^2)$, since we have to store all pairwise distances. This, of course, slows down running time, and even more importantly, prevents us from dealing with large graphs containing more than around 10,000 nodes. However, it appears that most graphs that can be drawn nicely possess much redundancy in the pairwise distances. We utilize this redundancy by neglecting most pairwise distances. Specifically, we define the set $\mathcal{S}$ of the "interesting" node pairs, so the stress energy is defined as:

$$\sum_{\{i,j\}\in\mathcal{S}} k_{ij} \left(|x_i - x_j| - d_{ij}\right)^2\,.$$

Accordingly, we can use a sparse Laplacian, whose nonzero entries correspond to the pairs in $\mathcal{S}$:

$$\mathcal{L}_{ij} = \begin{cases} -k_{ij} & \{i,j\} \in \mathcal{S} \\ \sum_{j:\{i,j\}\in\mathcal{S}} k_{ij} & i = j \\ 0 & \text{otherwise} \end{cases} \qquad i,j = 1,\ldots,n.$$

Similarly, the vector $b^{\tilde{x}}$ is defined as:

$$b_i^{\tilde{x}} = \sum_{\substack{j: \\ \{i,j\}\in\mathcal{S},\tilde{x}_j\leqslant\tilde{x}_i}} k_{ij}d_{ij} - \sum_{\substack{j: \\ \{i,j\}\in\mathcal{S},\tilde{x}_j>\tilde{x}_i}} k_{ij}d_{ij} \qquad i = 1,\ldots,n.$$

How can we choose the pairs in $\mathcal{S}$? In fact, such sparsification of the pairwise distances was already used in the multiscale stress-minimization methods of [4–6], where *in each scale* only the relations between nodes and their close neighborhood are considered. Here, we would like to offer another, very efficient sparsification scheme, which is especially appropriate for the new optimization method. An important feature of our sparsification is that it guarantees linear space requirements for bounded degree graphs. Moreover, using more sophisticated techniques that will be described in a forthcoming paper [10], we can maintain the linear space requirements for any kind of graph.

We construct the pairs-set, $\mathcal{S}$, to consist of two kinds of node-pairs: (1) "pivot-based" pairs and (2) pairs of close nodes. This construction is accomplished in the following way. First, we construct a set $\mathcal{P}$ that contains $m$ *pivot nodes* uniformly scattered over the graph. Such pivots can be chosen randomly. A better choice that we use in practice is described in [7]. We then initialize $\mathcal{S}$ with all the pairs $\{\{i,j\} \mid i \in P, j \in V, i \neq j\}$. The distances between these pairs preserve the overall structure of the graph. In [10] we describe a more advanced technique where these pivot-based pairs are enough. However, here, we should add to $\mathcal{S}$ all pairs that are close in the graph: $\{\{i,j\} \mid d_{ij} \leqslant B\}$. Typical values for $B$ are 3 to 6. This way we account for the local aesthetics of the drawing, which are not captured well by the pivots.

Computation of the distances from the pivots is performed in time $O(m \cdot |E|)$, and the storage requirements are $O(m \cdot |V|)$. We cannot state here a satisfactory recipe for finding an adequate value of $m$, and of course increasing $m$ never degrades the quality of the drawing. However, in our experiments, working with $30 \leqslant m \leqslant 100$ seems to provide good results. Consequently, our optimization process becomes efficient in terms of time and space, without affecting its drawing quality very much. Interestingly, such a sparsification approach completely fails with the node-by-node local stress optimization suggested by Kamada and Kawai, for reasons presently unknown. Anyway, this is an advantage of our new optimization method over the more traditional one.

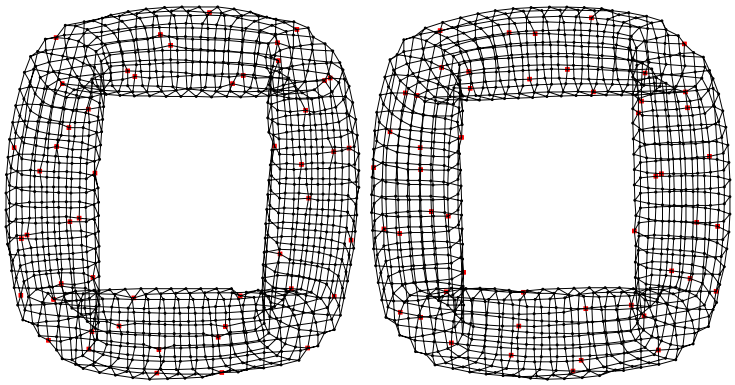
**Smoothing the pivots.** One minor problem may arise when working with pivots. Since the pivots are related to many more pairwise distances than the rest of the nodes, for some graphs they appear to "break out" of the drawing. For example, consider the 2-D layout of the $64 \times 16$ torus shown in Fig. 1(a). (Details about the computation of 2-D layouts are given in the following sections.) Here, we have set $B = 5$ and used 50 pivots, which are designated in the layout as large red squares. As the reader can see, the pivots are not very well-placed.

There is an easy solution to this problem. At a postprocessing stage, we fix the places of all the non-pivots, and then apply our algorithm only to the pivots, taking into account just the short distances (smaller than $B$). Consequently, in each iteration we have to solve a system of only $m$ equations. Furthermore, typically the pivots are not placed close together, so we can solve independently one equation for each of them, i.e., for each pivot node, $i$, we iteratively perform:

$$x_i \leftarrow \frac{\sum_{j\neq i:\ x_j\leqslant x_i, d_{ij}\leqslant B} k_{ij}(x_j + d_{ij}) + \sum_{j:\ x_j > x_i, d_{ij}\leqslant B} k_{ij}(x_j - d_{ij})}{\sum_{j\neq i:\ d_{ij}\leqslant B} k_{ij}}$$

until $x_i$ no longer changes.

All this local smoothing takes negligible time. Figure 1(b) shows again the layout of the $64 \times 16$ torus, but now after using the pivot-smoothing.



(a) (b)

**Fig. 1.** Drawing a $64 \times 16$ torus; pivots are the red squares. (a) Pivots seem to slightly break out of the drawing. (b) Pivot-smoothing is performed, so that pivots are better placed.

## 3 Working in the Presence of Predefined Coordinates

The main application of 1-D graph drawing is where one axis of the drawing is given in advance. Therefore, we should account for the already computed coordinates. Note that a careless computation that ignores the precomputed coordinates can be very problematic. Such a computation might yield new coordinates that are very similar to the given ones, resulting in a drawing whose intrinsic dimensionality would really be 1, meaning that one axis would be wasted.

Fortunately, we have found a very effective way of taking care of the predefined axis, which we term in this section axis $z$. The stress minimization strategy strives to achieve the target distances in the drawing. Some fraction of the target distances is already achieved in the predefined axis, $z$. Hence, when computing a new axis we would like the achieve the *residual target distances* instead of the original ones. The new residual target distances are defined as:

$$d_{ij}^z = \begin{cases} \sqrt{d_{ij}^2 - (z_i - z_j)^2} & d_{ij} > |z_i - z_j| \\ 0 & \text{otherwise} \end{cases} \qquad i, j = 1, \ldots, n\,. \tag{4}$$

Note that there is no reason to alter the normalization weights $k_{ij}$.

However, there is still one problem remaining: The target distances set the scale of the layout. Therefore, it is possible that the predefined axis, $z$, has an entirely different scale (e.g., it may be very small), so our computation of residual target distances makes no sense. To overcome this problem, we re-scale $z$ in order to bring it to the scale of the target distances (alternatively, we could re-scale the target distances in order to bring them to the scale of $z$). More specifically, we want to compute a constant $c > 0$ that minimizes the stress energy of the scaled $c \cdot z$. To find the exact value of $c$, we differentiate:

$$\frac{\partial}{\partial c} E(c \cdot z) = 0$$

The solution is:

$$c = \frac{\sum_{\{i,j\}\in\mathcal{S}} k_{ij} d_{ij} |z_i - z_j|}{\sum_{\{i,j\}\in\mathcal{S}} k_{ij} (z_i - z_j)^2} \,. \tag{5}$$

To summarize, when an axis $z$ is given in advance, we replace it with $c \cdot z$ to bring it into the scale of the target distances. The constant $c$ is computed by (5). Then, we take into account the distances already captured by $z$, by computing the residual target distances, $d^z_{ij}$, as in (4). The additional axis $x$ is computed in order to minimize the stress function $\sum_{\{i,j\}\in\mathcal{S}} k_{ij} \left(|x_i - x_j| - d^z_{ij}\right)^2$.

## 4 Experimental Results

### 4.1 Application for drawing digraphs

The digraph drawing algorithm in [1] computes the $y$-axis by minimizing the hierarchy energy in order to convey the overall directionality of the graph. The $x$-axis was computed by Eigen-projection or by minimum linear arrangement, and shows additional properties of the graph. The new 1-D stress minimization algorithm can serve in computing the $x$-axis. Our experiments show that in most cases using stress minimization improves the quality of the layouts by being able to accurately show sharp changes in the graph structure. Examples are shown in [2].

### 4.2 Two-dimensional layouts

Our 1-D stress minimization algorithm can be used to compute 2-D layouts. We first compute the $x$-coordinates, and then compute the $y$-coordinates given the $x$-coordinates, as explained in Sec. 3. Several results are shown in Fig. 2.

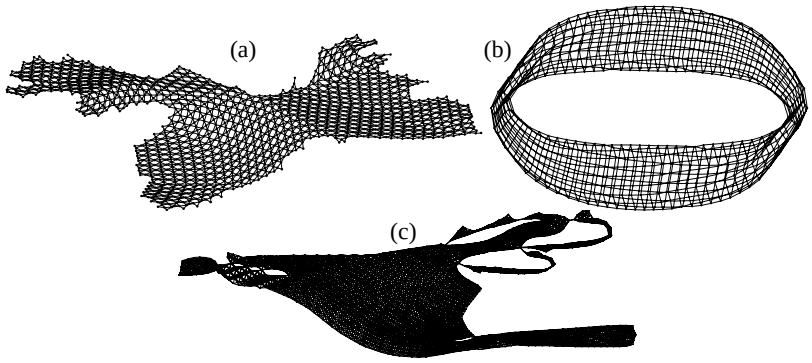



**Fig. 2.** 2-D layouts by 1-D stress minimization of: (a) Plsk1919, (b) $64 \times 16$ torus, (c) Shuttle.

Comparing the drawings in Fig. 2 to the drawings of the same graphs in Figs. 1, and 3(a,b) reveals that the results given in Fig. 2 are sub-optimal. Since the $x$-coordinates were computed independently of the $y$-coordinates, the algorithm strives to minimize the stress energy as much as possible using only the $x$-axis. This kind of a "greedy" approach stretches the layout along the $x$-axis. However, when we minimize the stress in 2-D from the beginning (using a method we describe in Sec. 5 or by employing the Kamada and Kawai algorithm [11]), the overall result is better. We provide quantitative results by comparing the values of the stress energy in the following table. For each of the three graphs we measured its stress energy twice: once only along the $x$-axis

and another time in the 2-D drawing. The results show that when computing the $x$-axis independently, the stress along it is smaller, but the overall 2-D stress is larger compared with a method that computes the two axes together. We have concluded that the algorithm as described so far is more suitable for achieving 1-D layouts.

| Graph name | Separate computation of axes | | Coupled computation of axes | |
|---|---|---|---|---|
| | $x$-axis stress | 2-D stress | $x$-axis stress | 2-D stress |
| **Plsk1919** | 198,343.09 | 66,367.90 | 347,724.76 | 20,141.41 |
| **Torus 64 $\times$ 16** | 107,855.51 | 43,581.09 | 144,607.29 | 26,029.12 |
| **Shuttle** | 529,776.08 | 203,817.77 | 1,035,382.73 | 48,909.45 |

### 4.3 Speed of Computation for Selected Graphs

*Smart initialization*

We recommend using our algorithm with a smart initialization. This way the probability of avoiding poor local minima is improved. Also, running time is significantly decreased as the number of iterations is reduced. Moreover, for a random initialization, we should enlarge the set of target distances $\mathcal{S}$ (by increasing the parameters $m$ or $B$), which further increases running time. Such a smart initialization could result from faster methods like those of [7, 8]. However, better results are obtained by initializing the algorithm by an approximation of itself that addresses a simplified problem. For such an approximation one can use the multiscale strategy and initialize the layout with a layout of a related, but much smaller graph. However, we have done it differently. We initialized the algorithm by first running a few iterations of the method described in the forthcoming paper [10], which accelerates the stress minimization by constraining the layout to lie in a unique small subspace. This significantly reduced the number of iterations, and allowed us to run the sparse version of the algorithm with parameters $m$=50 and $B$=3, yielding good results for all the given graphs. Note that sometimes increasing $B$ improves the quality of the layouts.

Table 1 shows the number of iterations for several graphs. For each graph we have computed a 2-D layout, so the results refer to the computation of two axes. The overall running time, which includes the smart initialization, was measured on a Pentium IV 2GHz PC with 256MB RAM. As shown, for a medium-sized graph (up to $\sim 10,000$ nodes), running times are comparable to those of the multi-scale stress minimization algorithms [4–6]. However, due to the better space complexity, our new method might deal with graphs of $\sim 100,000$ nodes.

| Graph name | $\|V\|$ | $\|E\|$ | x-axis iterations | y-axis iterations | Time (sec.) |
|---|---|---|---|---|---|
| **Torus 64 $\times$ 16** | 1024 | 2048 | 11 | 8 | 0.33 |
| **4970** | 4970 | 7400 | 19 | 22 | 3.66 |
| **Shuttle (Data)**† | 2851 | 15093 | 64 | 61 | 5.28 |
| **Crack**† | 10240 | 30380 | 41 | 27 | 15.95 |
| **Fidap006**§ | 1651 | 23914 | 49 | 23 | 2.42 |
| **Nos3**§ | 960 | 7442 | 20 | 12 | 0.58 |
| **Nos5**§ | 468 | 2352 | 18 | 20 | 0.27 |
| **Nos6**§ | 675 | 1290 | 10 | 4 | 0.14 |
| **Nos7**§ | 729 | 1944 | 15 | 29 | 0.36 |
| **Plat362**§ | 362 | 2712 | 9 | 5 | 0.13 |
| **Plat1919**§ | 1919 | 15240 | 17 | 31 | 1.73 |
| **Plsk1919**§ | 1919 | 4831 | 8 | 19 | 1.16 |
| **Sierpinski (depth 10)** | 88575 | 177147 | 40 | 22 | 165.64 |
| **Grid 317 $\times$ 317** | 100489 | 200344 | 17 | 5 | 189.66 |

† From Walshaw's collection: `www.gre.ac.uk/~c.walshaw/partition`

§ From the Matrix Market collection: `math.nist.gov/MatrixMarket`

**Table 1.** Number of iterations and running time (in seconds) for various graphs

## 5 Alternating Two-Dimensional Stress Minimization

As we have seen, directly applying our algorithm to compute a 2-D layout produces imperfect results, in which the graph is often "stretched" along the first computed axis. However, we can overcome this artifact by coupling the computation of the two axes, thereby neutralizing any preference for one of them. Here is the precise algorithm:

**Function 2-D_stress_minimization** ($G(V,E),\ x,y \in \mathbb{R}^n$)
% Coordinates of node $i$ are $(x_i, y_i)$
  Compute the Laplacian $\mathcal{L}$
  **do**
    $\tilde{x} \leftarrow x$
    Compute $b^{\tilde{x}}$ using the residual distances $d_{ij}^{y}$
    Compute $x$ for which $\mathcal{L}x = b^{\tilde{x}}$
    $\tilde{y} \leftarrow y$
    Compute $b^{\tilde{y}}$ using the residual distances $d_{ij}^{x}$
    Compute $y$ for which $\mathcal{L}y = b^{\tilde{y}}$
  **while** ($x \neq \tilde{x}$ or $y \neq \tilde{y}$)

This technique can be directly extended to higher dimensions. This way, we can use our algorithm to compute multidimensional layouts. The results are comparable to the Kamada-Kawai algorithm, but there is the advantage of the fast, pivot-based algorithm, whose memory requirements are lower than other implementations of the Kamada-Kawai algorithm. In [10] we describe another advantage of this technique, which enables an additional substantial reduction of time and space complexity.

The actual running times of the algorithm are slightly larger than the results given in Table 1 (mainly because of the added time for recomputing the residual distances at each iteration). Results are shown in Fig. 3 (and in Fig. 1).

## 6 Discussion

Strategies based on minimizing the stress energy are known for their ability to produce nice graph layouts. Common optimization processes are based on node-by-node local optimization methods. Specifically, Kamada and Kawai [11], use a localized 2-dimensional Newton-Raphson process. However, working with a full Newton-Raphson process is impractical, since it requires a recomputation of the Hessian matrix in each step. Moreover, the Hessian might be singular or ill-conditioned, which may cause additional numerical difficulties. We have devised an algorithm for minimizing the stress energy needed for computing 1-D layouts. Technically, our algorithm introduces a rather global optimization process that replaces the usual local node-by-node optimization. Interestingly, it can be proved that this process is equivalent to performing a full $n$-dimensional Newton-Raphson optimization, unlike the local 2-dimensional process used by Kamada and Kawai. Hence, while the common wisdom is that stress-minimization is very difficult for 1-D layouts as optimization processes tend to become stuck in poor local minima, we have shown that the 1-D case is indeed the easier case allowing the use of a global optimization process. This motivates us to try to find ways to incorporate the powerful 1-D optimization process for computing multidimensional layouts. Consequently, we devised an algorithm for axis-by-axis computation of multi-dimensional layouts that has an advantage over other stress minimization approaches in terms of time and space complexity.

## References

1. L. Carmel, D. Harel and Y. Koren, "Drawing Directed Graphs Using One-Dimensional Optimization", *Proc. Graph Drawing 2002*, LNCS 2528, pp. 193–206, Springer-Verlag, 2002.

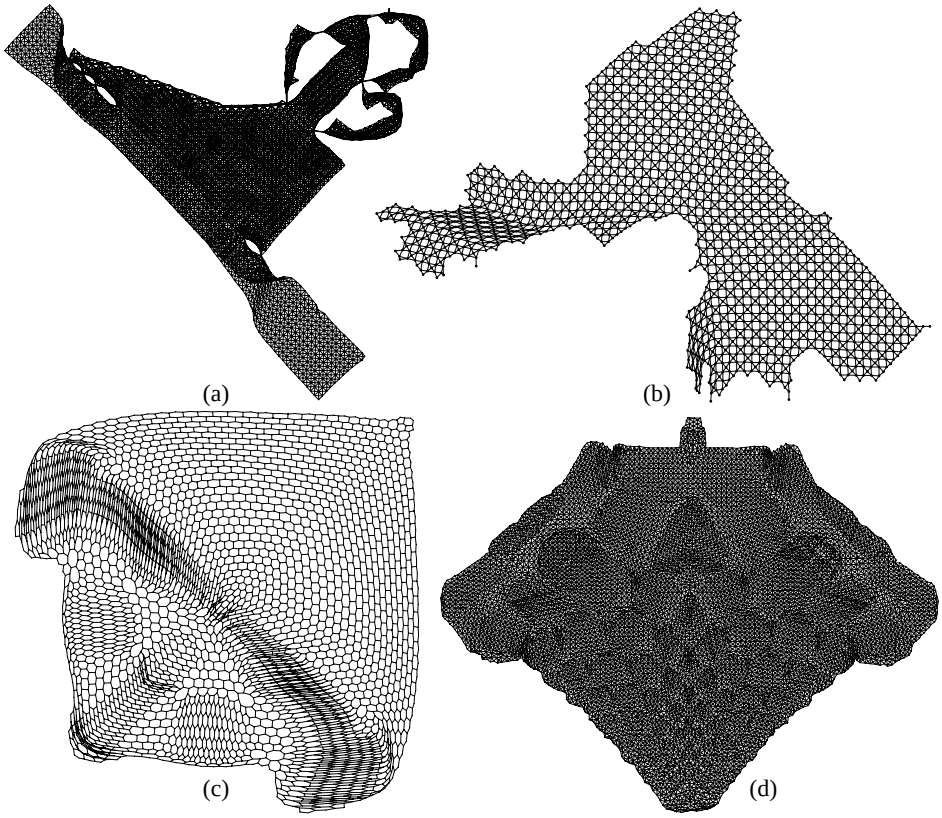



**Fig. 3.** Drawing by alternating stress minimization of: (a) Shuttle, (b) Plsk1919, (c) 4970, (d) Crack.


2. L. Carmel, D. Harel and Y. Koren, "Combining Hierarchy and Energy for Drawing Directed Graphs", *IEEE Transactions on Visualization and Computer Graphics*, IEEE, in press.
3. J. D. Cohen, "Drawing Graphs to Convey Proximity: an Incremental Arrangement Method", *ACM Transactions on Computer-Human Interaction* **4** (1997), 197–229 .
4. P. Gajer, M. T. Goodrich and S. G. Kobourov, "A Multi-dimensional Approach to Force-Directed Layouts of Large Graphs", *Proc. Graph Drawing 2000*, LNCS 1984, pp. 211–221, Springer-Verlag, 2000.
5. R. Hadany and D. Harel, "A Multi-Scale Method for Drawing Graphs Nicely", *Discrete Applied Mathematics* **113** (2001), 3-21.
6. D. Harel and Y. Koren, "A Fast Multi-Scale Method for Drawing Large Graphs", *Journal of Graph Algorithms and Applications* **6** (2002), 179–202.
7. D. Harel and Y. Koren, "Graph Drawing by High-Dimensional Embedding", *Proc. Graph Drawing 2002*, LNCS 2528, pp. 207–219, Springer-Verlag, 2002.
8. Y. Koren, L. Carmel and D. Harel, "ACE: A Fast Multiscale Eigenvectors Computation for Drawing Huge Graphs", *Proc. IEEE Information Visualization (InfoVis'02)*, IEEE, pp. 137–144, 2002.
9. Y. Koren and D. Harel, "One-Dimensional Graph Drawing: Part I — Drawing Graphs by Axis Separation", Technical report MCS03-08, Faculty of Math. and Computer Science, The Weizmann Institute of Science, 2003.
10. Y. Koren, "Graph Drawing by Subspace Optimization", to be published.
11. T. Kamada and S. Kawai, "An Algorithm for Drawing General Undirected Graphs", *Information Processing Letters* **31** (1989), 7–15.
12. J. W. Sammon, "A Nonlinear Mapping for Data Structure Analysis", *IEEE Trans. on Computers* **18** (1969), 401–409.