

Graph Drawing by High-Dimensional Embedding

David Harel

Dept. of Computer Science and Applied Mathematics
The Weizmann Institute of Science, Rehovot, Israel
<http://www.wisdom.weizmann.ac.il/~dharel/>
dharel@wisdom.weizmann.ac.il

Yehuda Koren

AT&T Labs – Research
Florham Park, NJ 07932
<http://www.research.att.com/~yehuda/>
yehuda@research.att.com

Abstract

We present a novel approach to the aesthetic drawing of undirected graphs. The method has two phases: first embed the graph in a very high dimension and then project it into the 2-D plane using principal components analysis. Running time is linear in the graph size, and experiments we have carried out show the ability of the method to draw graphs of 10^5 nodes in few seconds. The new method appears to have several advantages over classical methods, including a significantly better running time, a useful inherent capability to exhibit the graph in various dimensions, and an effective means for interactive exploration of large graphs.

Article Type	Communicated by	Submitted	Revised
Regular Paper	Xin He	March 2003	November 2003

1 Introduction

A graph $G(V = \{1, \dots, n\}, E)$ is an abstract structure that is used to model a relation E over a set V of entities. Graph drawing is a standard means for the visualization of relational information, and its ultimate usefulness depends on the readability of the resulting layout; that is, the drawing algorithm's capability of conveying the meaning of the diagram quickly and clearly. Consequently, many approaches to graph drawing have been developed [4, 15]. We concentrate on the problem of drawing undirected graphs with straight-line edges, and the most popular approaches to this appear to be those that define a cost function (or a force model), whose minimization determines the optimal drawing. The resulting algorithms are known as *force-directed* methods [1, 3, 5, 8, 14].

We suggest a new approach to the problem of graph drawing, relying on the observation that laying out a graph in a high dimension is significantly easier than drawing it in a low dimension. Hence, the first step of our algorithm is to quickly draw the graph in a very high dimensional space (e.g., in 50 dimensions). Since standard visualization techniques allow using only 2 or 3 dimensions, the next step of our algorithm is to algorithmically project the high-dimensional drawing into a low dimension. For this, we are using a well-known multivariate analysis technique called *principal components analysis* (PCA).

The resulting algorithm is extremely fast, yet very simple. Its time complexity is $O(m \cdot |E| + m^2 \cdot n)$, where m is the dimension in which the graph is embedded during the first stage of the algorithm. In fact, the running time is linear in the graph's size, since m is independent of it. Typical computation times are of less than 3 seconds for 10^5 -node graphs, and are thus significantly faster than force-directed approaches. As to the quality of the drawings, Section 4 shows several very encouraging results.

Beside the main application of our work, which is drawing large graphs, another possible application is in finding the best viewpoint for projecting a 3-D graph drawing into the plane; see Subsection 6.1.

2 Drawing Graphs in High Dimension

Frequently, drawing a graph so as to achieve a certain aesthetic criterion cannot be optimally achieved in a low dimension, due to the fact that several aesthetic goals have to compete on a shared limited space. Thus, being able to carry out the initial drawing work in many dimensions leaves more space for richer expression of the desired properties, and thus makes the entire task easier.

For example, consider the task of drawing a graph consisting of a single cycle. For any reasonable aesthetical consideration, such a cycle can be drawn optimally in two or more dimensions, but cannot be drawn well in one dimension. More generally, when confronting the task of embedding a graph in an Euclidean space (of any dimension) so as to equate the graph-theoretical node-node dis-

tances and the Euclidean node-node distances¹ quite often two dimensions are not enough. (For example, consider embedding a torus or a cube in 2-D.) In fact, drawing a graph in more than two dimensions is already a familiar technique; see, e.g., [1, 9, 15]. In many cases it was found that the higher dimensional drawing can improve the quality of the drawing in a fundamental way.

Our method for constructing a multidimensional layout is straightforward. In order to draw a graph in m dimensions, we choose m *pivot* nodes that are almost uniformly distributed on the graph and associate each of the m axes with a unique node. Axis i , which is associated with pivot node p_i , represents the graph from the “viewpoint” of node p_i . This is done by defining the i -th coordinate of each of the other nodes as its graph-theoretic distance from p_i . Hence p_i is located at place 0 on axis i , its immediate neighbors are located at place 1 on this axis, and so on.

More formally, denote by d_{uv} the graph-theoretic distance between node v and node u . Let *Pivots* be some set $\{p_1, p_2, \dots, p_m\} \subset V$. Each node $v \in V$ is associated with m coordinates $X^1(v), X^2(v), \dots, X^m(v)$, such that $X^i(v) = d_{p_i v}$.

The resulting algorithm for drawing the graph in m dimensions is given in Fig. 1. The graph theoretic distances are computed using breadth-first-search (BFS). (When edges are positively weighted, BFS should be replaced by Dijkstra’s algorithm; see e.g., [2].) The set *Pivots* is chosen as follows. The first member, p_1 , is chosen at random. For $j = 2, \dots, m$, node p_j is a node that maximizes the shortest distance from $\{p_1, p_2, \dots, p_{j-1}\}$. This method is mentioned in [12] as a 2-approximation² to the k -center problem, where we want to choose k vertices of V , such that the longest distance from V to these k centers is minimized. However, different approaches to selecting the pivots may also be suitable.

The time complexity of this algorithm is $O(m \cdot (|E| + |V|))$, since we perform BFS in each of the m iterations. A typical value of m is 50.

Here now are two observations regarding the properties of the resulting drawing. First, for every two nodes v and u and axis $1 \leq i \leq m$, we have:

$$|X^i(v) - X^i(u)| \leq d_{uv}$$

This follows directly from the triangle inequality, since:

$$\begin{aligned} d_{p_i u} &\leq d_{p_i v} + d_{uv} \text{ and } d_{p_i v} \leq d_{p_i u} + d_{uv} \\ \implies |X^i(v) - X^i(u)| &= |d_{p_i v} - d_{p_i u}| \leq d_{uv} \end{aligned}$$

Thus, nodes that are closely related in the graph will be drawn close together.

In order to get a nice layout we also have to guarantee the opposite direction, i.e., that non-adjacent nodes are not placed closely. This issue, which is handled

¹Such a distance preserving embedding possess a tight connection to aesthetically pleasing layout, as reflected in the well-known graph drawing algorithm of Kamada and Kawai [14]

²A δ -approximation algorithm delivers an approximate solution guaranteed to be within a constant factor δ of the optimal solution.

```

Function HighDimDraw ( $G(V = \{1, \dots, n\}, E), m$ )
% This function finds an  $m$ -dimensional layout of  $G$ :

Choose node  $p_1$  randomly from  $V$ 
 $d[1, \dots, n] \leftarrow \infty$ 
for  $i = 1$  to  $m$  do
  % Compute the  $i$ -th coordinate using BFS
   $d_{p_i^*} \leftarrow \text{BFS}(G(V, E), p_i)$ 
  for every  $j \in V$ 
     $X^i(j) \leftarrow d_{p_i^* j}$ 
     $d[j] \leftarrow \min\{d[j], X^i(j)\}$ 
  end for
  % Choose next pivot
   $p_{i+1} \leftarrow \arg \max_{\{j \in V\}} \{d[j]\}$ 
end for
return  $X^1, X^2, \dots, X^m$ 

```

Figure 1: Drawing a graph in m dimensions

mainly by the projection part of the algorithm (discussed in the next section), brings us to the second observation. Here we observe a kind of separation between nodes that are distant in the graph.

For an axis i and nodes u and v , denote $\delta_{v,u}^i \stackrel{\text{def}}{=} \min\{d_{p_i v}, d_{p_i u}\}$. Then, for every $v, u \in V$ and axis $1 \leq i \leq m$, we have:

$$|X^i(v) - X^i(u)| \geq d_{uv} - 2\delta_{v,u}^i$$

For the proof, assume w.l.o.g. that $\delta_{v,u}^i = d_{p_i v}$. Again, using the triangle inequality:

$$\begin{aligned} d_{uv} &\leq d_{p_i v} + d_{p_i u} = d_{p_i v} + d_{p_i v} + (d_{p_i u} - d_{p_i v}) = 2\delta_{v,u}^i + |X^i(v) - X^i(u)| \\ \implies d_{uv} - 2\delta_{v,u}^i &\leq |X^i(v) - X^i(u)| \end{aligned}$$

Thus if we denote the minimal distance between $\{v, u\}$ and *Pivots* by:

$$\epsilon_{v,u} \stackrel{\text{def}}{=} \min_{i \in \{1, \dots, m\}, j \in \{v, u\}} d_{p_i j},$$

then there exists an axis i such that $|X^i(v) - X^i(u)| \geq d_{uv} - 2\epsilon_{v,u}$.

Since we have chosen the pivots in order to minimize their distance to the rest nodes, we expect $\epsilon_{v,u}$ to be fairly small.

3 Projecting Into a Low Dimension

At this stage we have an m -dimensional drawing of the graph. In order to visually realize the drawing we have to project it into 2 or 3 dimensions. Picking a

good projection is not straightforward, since the axes are correlated and contain redundant information. Moreover, several axes may scatter nodes better than others, thus being more informative. For example, consider a square grid. If we use two axes that correspond to two opposite corners, the resulting drawing will be essentially 1-dimensional, as the two axes convey basically the same information and are anti-correlated. (That is, being “near” one corner is exactly like being “far” from the opposite corner.) Also, taking an axis associated with a boundary node is very often more informative than taking an axis associated with a central node; the first case causes the nodes to be scattered in a much better way, since the maximal distance from a boundary node is about twice as large as the maximal distance from a central node.

To address these issues we use a tool that is well known and in standard use in multivariate analysis — *principal component analysis* (PCA). PCA transforms a number of (possibly) correlated variables into a (smaller) number of uncorrelated variables called principal components (PCs). The first principal component accounts for as much of the variability in the data as possible, and each succeeding component accounts for as much of the remaining variability as possible. By using only the first few principal components, PCA makes it possible to reduce the number of significant dimensions of the data, while maintaining the maximum possible variance thereof. See [7] for a comprehensive discussion of PCA.

In our case, we have m n -dimensional variables X^1, \dots, X^m , describing the n nodes in m dimensions. We want to represent the n nodes using only k dimensions (typically $k = 2$), using k n -dimensional *uncorrelated* vectors Y^1, \dots, Y^k , which are the principal components. Hence, the coordinates of node i are $(Y^1(i), \dots, Y^k(i))$. Each of the PCs among Y^1, \dots, Y^k is a linear combination of the original variables X^1, \dots, X^m .

Here are the details. Denote the mean of i -th axis by $m_i \stackrel{\text{def}}{=} \sum_{j=1}^n \frac{X^i(j)}{n}$. The first stage of the PCA is to center the data around 0 which is just a harmless translation of the drawing. We denote the vectors of centered data by $\hat{X}^1, \dots, \hat{X}^m$, defined as:

$$\hat{X}^i(j) = X^i(j) - m_i, \quad i = 1, \dots, m, \quad j = 1, \dots, n$$

We now construct an $m \times n$ matrix, X , whose rows are the (centered) coordinates:

$$X = \begin{pmatrix} \hat{X}^1(1) & \dots & \dots & \hat{X}^1(n) \\ \vdots & \vdots & \vdots & \vdots \\ \hat{X}^m(1) & \dots & \dots & \hat{X}^m(n) \end{pmatrix}$$

The *covariance matrix* S , of dimension $m \times m$, is defined as

$$S = \frac{1}{n} X X^T$$

We now have to compute the first k eigenvectors of S (those that correspond to the largest eigenvalues). We denote these eigenvectors by u_1, \dots, u_k . The vector

lengths should be normalized to 1, so that these k vectors are orthonormal. A simple method for computing the eigenvectors is described below.

Now to the projection itself. The first new axis, Y^1 , is the projection of the data in the direction of u_1 , the next axis, Y^2 , is the projection in the direction of u_2 , and so on. Hence the new coordinates are defined by:

$$Y^i = X^T u_i, \quad i = 1, \dots, k$$

For the interested reader, we now briefly discuss the theoretical reasoning behind the PCA process. The projection of the data in a certain direction can be formulated by $y = X^T u$, where u is a unit vector ($\|u\|_2 = 1$) in the desired direction. Since the original data is centered, the projection, y , is also centered. Thus, the variance of y can be written simply as $y^T y/n$. Note that,

$$\frac{1}{n} y^T y = \frac{1}{n} (X^T u)^T X^T u = \frac{1}{n} u^T X X^T u = u^T S u .$$

Consequently, to find the projection that retains the maximum variance, we have to solve the following constrained maximization problem:

$$\begin{aligned} \max_u u^T S u & \quad (1) \\ \text{subject to: } \|u\|_2 &= 1 \end{aligned}$$

Standard linear algebra shows that the maximizer of problem 1 is u_1 , the first eigenvector of S . Hence, Y^1 is the 1-dimensional projection of the data that has the maximal variance (i.e., in which the data is most scattered). Using similar techniques it can be shown that Y^1, \dots, Y^k constitute the k -dimensional projection of the data that yields the maximal variance. Moreover, the orthogonality of u_1, \dots, u_k implies $(Y^i)^T Y^j = 0$ for $i \neq j$. Hence, these k axes are uncorrelated.

In general, as we shall see in Section 4, it suffices to draw a graph on the plane using Y^1 and Y^2 only, thus scattering the nodes in a maximal fashion.³ However, sometimes using Y^3 or Y^4 may be useful too.

Regarding time complexity, the most costly step is computing the covariance matrix $S = \frac{1}{n} X X^T$. (In practice we do not divide by n , since multiplication by a constant does not change the eigenvectors.) This matrix multiplication is carried out in a straightforward way using exactly $m^2 n$ multiplications and additions, so the time complexity is $O(m^2 n)$, with a very small hidden constant (although matrix multiplication can be done faster in theory; see e.g., [2]).

As to computing the first eigenvectors of the $m \times m$ covariance matrix S (i.e., those that correspond to the largest eigenvalues), we use the simple power-iteration method; see e.g., [22]. Since $m \ll n$, the running time is negligible (taking in practice less than a millisecond) and there is no need for more complicated techniques. The basic idea is as follows. Say we are given an

³Thus, using PCA is, in a sense, incorporating a global “repulsive force”, in the terms used in force-directed methods.

$m \times m$ symmetric matrix A with eigenvectors u_1, u_2, \dots, u_m , whose corresponding eigenvalues are $\lambda_1 > \lambda_2 > \dots > \lambda_m \geq 0$. Let $x \in \mathbb{R}^n$. If x is not orthogonal to u_1 (i.e., $x^T u_1 \neq 0$) then the series Ax, A^2x, A^3x, \dots converges in the direction of u_1 . More generally, in the case where $x^T u_1 = 0, x^T u_2 = 0, \dots, x^T u_{j-1} = 0, x^T u_j \neq 0$, the series Ax, A^2x, A^3x, \dots converges in the direction of u_j . The full algorithm is depicted in Fig. 2.

```

Function PowerIteration ( $S - m \times m$  matrix )
% This function computes  $u_1, u_2, \dots, u_k$ , the first  $k$  eigenvectors of  $S$ .
const  $\epsilon \leftarrow 0.001$ 
for  $i = 1$  to  $k$  do
   $\hat{u}_i \leftarrow$  random
   $\hat{u}_i \leftarrow \frac{\hat{u}_i}{\|\hat{u}_i\|}$ 
  do
     $u_i \leftarrow \hat{u}_i$ 
    % orthogonalize against previous eigenvectors
    for  $j = 1$  to  $i - 1$  do
       $u_i \leftarrow u_i - (u_i^T u_j)u_j$ 
    end for
     $\hat{u}_i \leftarrow Su_i$ 
     $\hat{u}_i \leftarrow \frac{\hat{u}_i}{\|\hat{u}_i\|}$  % normalization
  while  $\hat{u}_i^T u_i < 1 - \epsilon$  % halt when direction change is negligible
   $u_i \leftarrow \hat{u}_i$ 
end for
return  $u_1, u_2, \dots, u_k$ 

```

Figure 2: The power iteration algorithm

4 Examples

Our algorithm was implemented in C, and runs on a dual processor Intel Xeon 1.7Ghz PC. Since the implementation is non-parallel, only one of the processors is used. For all the results given here we have set $m = 50$, meaning that the graphs are embedded in 50 dimensions. Our experience is that the results are not sensitive to the exact value of m . In fact, increasing m does not degrade the quality of the results, but doing so seems not to be needed. On the other hand, picking an overly small value for m may harm the smoothness of the drawing. We speculate that as the graphs are to be drawn in only two or three dimensions, a vast increase of m cannot be helpful.

Table 1 gives the actual running times of the algorithm on graphs of different sizes. In addition to the total computation time, we show the times of the two most costly parts of the algorithm — computing the m -dimensional embedding (Fig. 1) and computing the covariance matrix S . We want to stress the fact that since the algorithm does not incorporate an optimization process, the running

time is determined completely by the size of the graph (i.e., $|V|$ and $|E|$), and is independent of the structure of the graph. This is unlike force-directed methods.

Table 1: Running time (in seconds) of the various components of the algorithm. We denote with T_E the time to compute the high-dimensional embedding, with T_C the time to compute the covariance matrix and with T the total running time.

graph	$ V $	$ E $	T	T_E	T_C
516 [21]	516	729	0.00	0.00	0.00
Fidap006 [§]	1651	23,914	0.03	0.02	0.01
4970 [21]	4970	7400	0.08	0.03	0.05
3elt [†]	4720	13,722	0.09	0.05	0.05
Crack [‡]	10,240	30,380	0.30	0.14	0.08
4elt2 [†]	11,143	32,818	0.25	0.16	0.09
Sphere [†]	16,386	49,152	0.81	0.47	0.16
Fidap011 [§]	16,614	537,374	0.75	0.59	0.13
Sierpinski (depth 10)	88,575	177,147	1.77	0.89	0.77
grid 317 × 317	100,489	200,344	2.59	1.59	0.89
Ocean [†]	143,437	409,593	7.16	5.74	1.25
mrngA [†]	257,000	505,048	13.09	10.66	2.19
grid 1000 × 1000	1,000,000	1,998,000	50.52	41.03	8.48
mrngB [†]	1,017,253	2,015,714	57.81	47.83	8.84

[§] Taken from the Matrix Market, at:

<http://math.nist.gov/MatrixMarket>

[†] Taken from the University of Greenwich Graph Partitioning Archive, at:

<http://www.gre.ac.uk/~c.walshaw/partition>

[‡] Taken from Jordi Petit’s collection, at:

<http://www.lsi.upc.es/~jpetit/MinLA/Experiments>

Graphs of around 10^5 nodes take only a few seconds to draw, and 10^6 -node graphs take less than a minute. Thus, the algorithm exhibits a truly significant improvement in computation time for drawing large graphs over previously known ones.⁴

Following is a collection of several drawings produced by the algorithm. The layouts shown in Fig. 3 are typical results of our algorithm, produced by taking the first two principal components as the axes. In Fig. 3(a) we show a square grid with and in Fig. 3(b) we are showing the same grid with $\frac{1}{3}$ of the edges omitted at random. Figure 3(b) shows a folded grid, obtained by taking a square grid and connecting opposing corners. This graph has high level of symmetry, which is nicely reflected in the drawing. In Fig. 3(d) we are showing a torus.

⁴Our recent ACE algorithm, [18], exhibits similar advantages using totally different methods.

Figures 3(c,d) show two finite element graphs, whose drawings give a feeling of a 3-D landscape.

Sometimes it is aesthetically better to take different principal components. For example, in Fig. 4(a) the 516 graph is depicted using the first and second PCs, while in Fig. 4(b) the first and third PCs are used. Note that the second PC scatters the nodes better than the third PC, as must be the case. However, here, using the third PC instead of the second one results in an aesthetically superior drawing. A similar example is given in Fig. 4(c,d) with the Fidap006 graph. In fact, this is the typical case with many graphs whose nice drawing has an unbalanced aspect ratio. The first two axes provide a well balanced drawing, while using different axes (the third or the fourth PCs) yields a prettier result.

In fact, the algorithm also produces more information than others, by drawing the graph in a high dimension. Thus, we can view the graph from different viewpoints that may reveal interesting aspects of the graph. This is demonstrated in the drawing of the Sphere graph. Fig. 5(a) shows a drawing using the first and second PCs. The six “smooth” corners appearing in Fig. 5(a) become really salient in Fig. 5(b), where the third and fourth PCs are used. These corners are still clearly visible using the fourth and fifth PCs in Fig. 5(c), where a flower shape emerges.

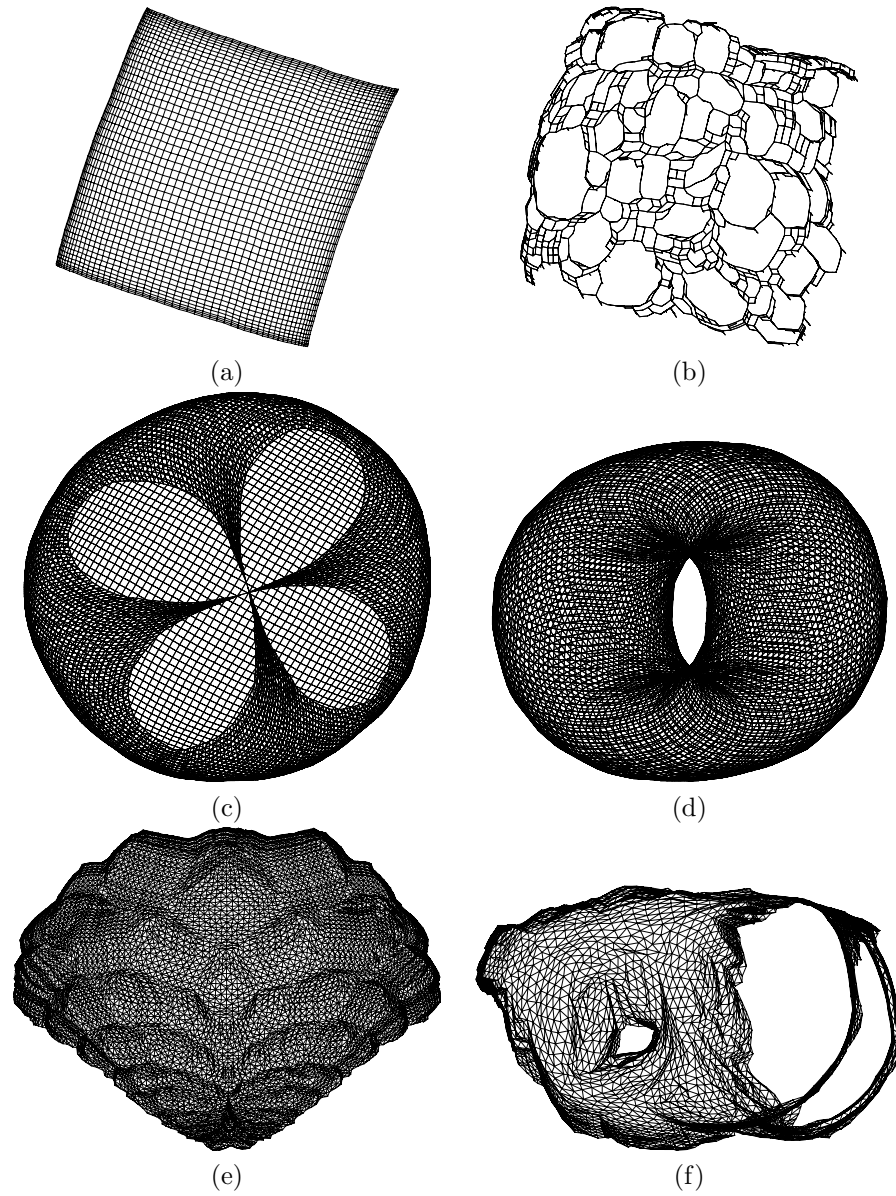


Figure 3: Layouts of: (a) A 50×50 grid; (b) A 50×50 grid with $\frac{1}{3}$ of the edges omitted at random; (c) A 100×100 grid with opposite corners connected; (d) A 100×100 torus; (e) The Crack graph; (f) The 3elt graph

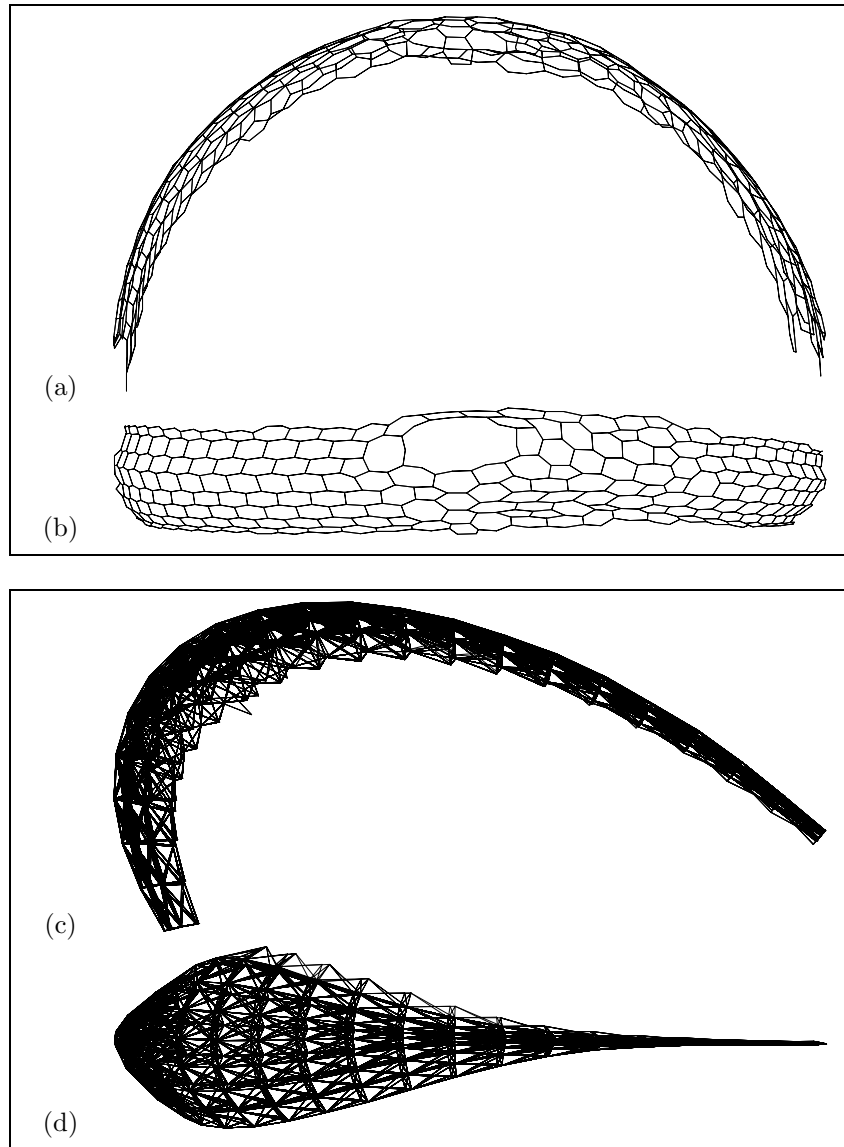


Figure 4: (a,b592) Drawing the 516 graph using: (a) 1st and 2nd PCs; (b) 1st and 3rd PCs. (c,d) Drawing the Fidap006 graph using: (c) 1st and 2nd PCs; (d) 1st and 3rd PCs

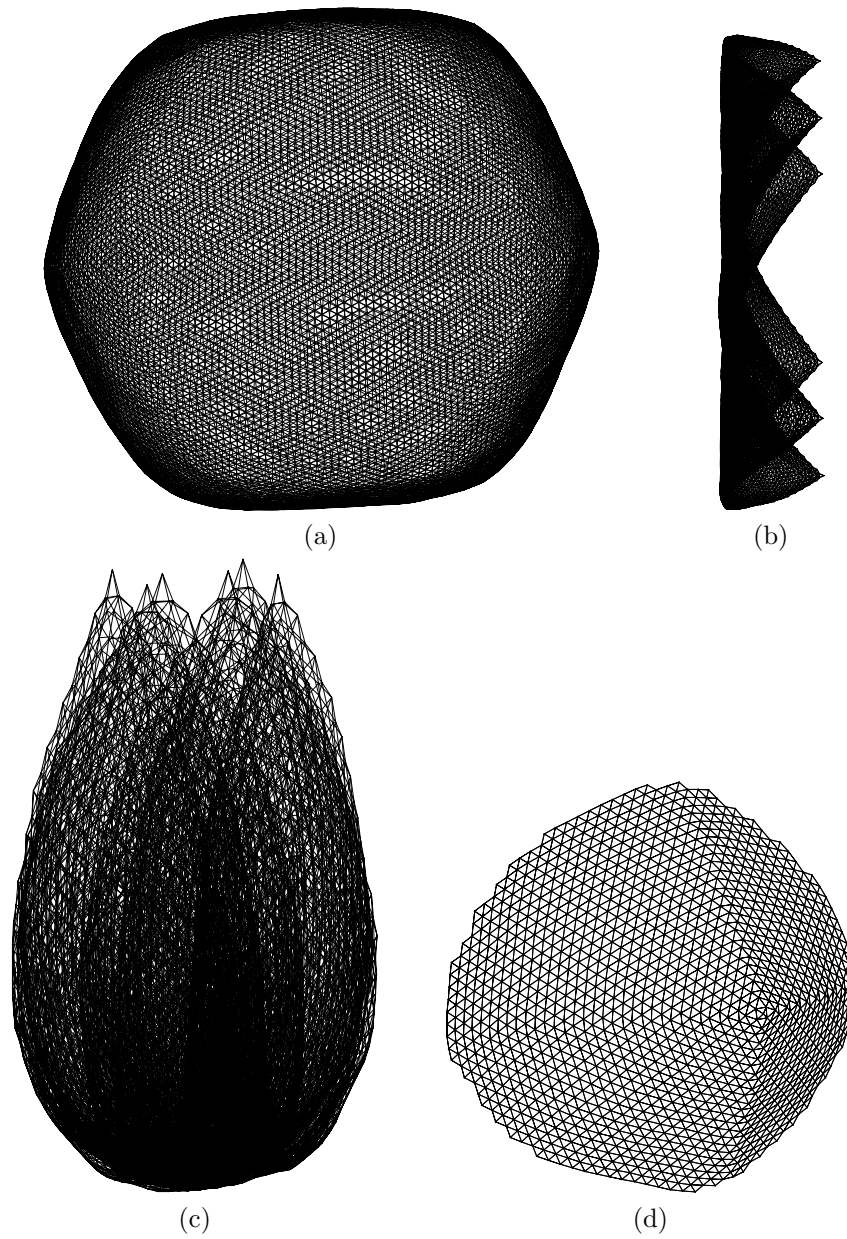


Figure 5: Multiple viewpoints of the Sphere graph: (a) first and second PCs; (b) third and fourth PCs; (c) fourth and fifth PCs; (d) zooming in on one of the corners

4.1 Zooming in on regions of interest

Drawings in two dimensions reveal only part of the richness of the original high dimensional drawing. Indeed, the 2-D drawing must forgo showing some properties of small portions of the graph, in order to get a well balanced picture of the entire graph. This facilitates a novel kind of interactive exploration of the graph structure: The user can choose a region of interest in the drawing and ask the program to zoom in on it. We then utilize the fact that we have a high dimensional drawing of the graph, which possibly contains a better explanation for the chosen subgraph than what shows up in 2-D. First we take the coordinates of the subgraph from the already computed m -dimensional drawing. We then use PCA to project these coordinates into 2-D. In this way we may reveal properties appearing in the high-dimensional drawing, which are not shown in the low-dimensional drawing of the full graph.

For example, we wanted to investigate the “corners” of the Sphere graph. We zoomed in on one of the corners, and the result is shown in Fig. 5(d). It can be seen that the corner is a meeting point of four faces. Another example is the dense graph, Fidap011, depicted in Fig. 6. Due to file size limitation, we cannot print this huge graph with adequate visual quality. Hence, it is very instructive to see parts of its micro-structure, as shown in the bottom of Fig. 6(b).

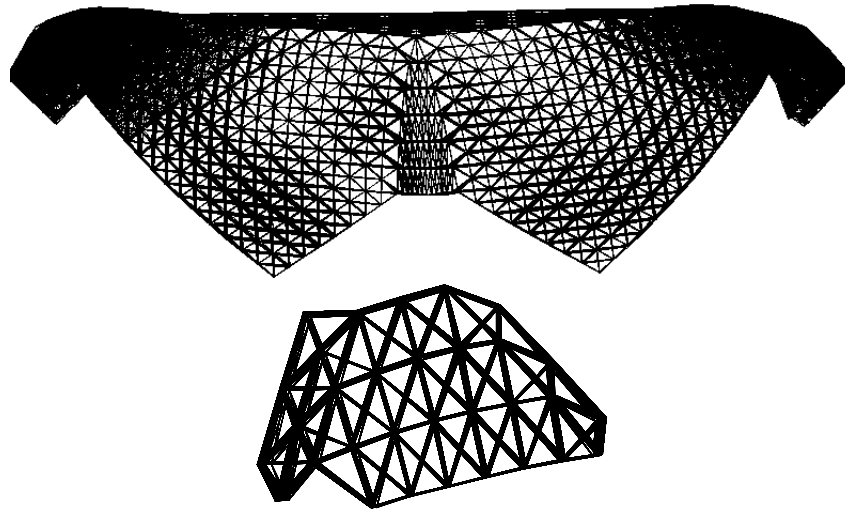


Figure 6: Top: The Fidap011 graph; Bottom: zooming in on the micro-structure

Additional related examples are given in Fig. 7. The Sierpinski fractal of depth 7, is shown in Fig. 7(a). Note that the left and top parts of it are distorted (in fact, they are explained by the third PC). In Fig. 7(b) we depict the result of zooming-in on the left part of the graph, revealing its nice structure. The layout of the 4elt2 graph, depicted in Fig. 7(c), resembles the one obtained by [18]. For a better understanding of its structure we may zoom-in on parts of the

drawing. Fig. 7(d) shows the results of zooming-in on the bottom strip. In Fig. 7(e) we provide a drawing of the Ocean graph, containing over 143,000 nodes. To understand its micro-structure we zoom-in on it, providing a sample result in Fig. 7(f). The last example is the 4970 graph, nicely depicted in Fig. 7(g). We zoom-in on its top-center portion, as shown in Fig. 7(h).

Before ending this section, we should mention that our algorithm is not suitable for drawing trees; see e.g. Fig. 8. In fact, for tree-like graphs, it may be very hard to pick a suitable viewpoint for projection. This is probably due to the fact that the high dimensional drawing of these graphs spans a “wild” subspace of quite a high dimension. Moreover, if there is no pivot within some subtree, all the nodes that have the same distance to the root of the subtree (the node that connects the subtree to the rest of the graph) will get exactly the same coordinates in the high-dimensional embedding.⁵

5 Related Work

The most common approach to drawing undirected graphs with straight line edges is based on defining a cost function (or a force model), whose minimization determines the optimal drawing. Such techniques are known as *force-directed* methods [4, 15].

In terms of performance and simplicity, the algorithm has some significant advantages when compared to force-directed methods. To appreciate these advantages, let us make a short divergence for surveying the state of the art in force-directed drawing of large graphs. A naive implementation of a force-directed method encounters real difficulties when dealing with graphs of more than a few hundred nodes. These difficulties stem from two reasons. First, in a typical force model there is a quadratic number of forces, making a single iteration of the optimization process very slow. Second, for large graphs the optimization process needs too many iterations for turning the initial random placement into a nice layout. Some researchers [19, 20] have improved these methods to some extent, by accelerating force calculation using *quad-trees* that reduce the complexity of the force model. This way, [20] reports drawing 1000-node graphs in around 5 minutes and [19] mentions vastly improved running times per a single iteration. Whereas using quad-trees addresses the first issue by accelerating each single iteration, there is still the second issue of getting out of the initial random placement. Both these issues receive adequate treatment by incorporating the *multi-scale strategy* as suggested by several authors; see [9, 10, 11, 21]. These methods considerably improve running times by rapidly constructing a simplified initial globally nice layout and then refining it locally. The fastest of them all, [21], draws a 10^5 -node graph in a typical time of ten minutes. Coming back to our algorithm, we do not have an optimization process, so we do not encounter the aforementioned difficulties of the force-directed approach. Our algorithm is considerably faster than all of these, however, be-

⁵This observation was brought to our attention by Ulrik Brandes.

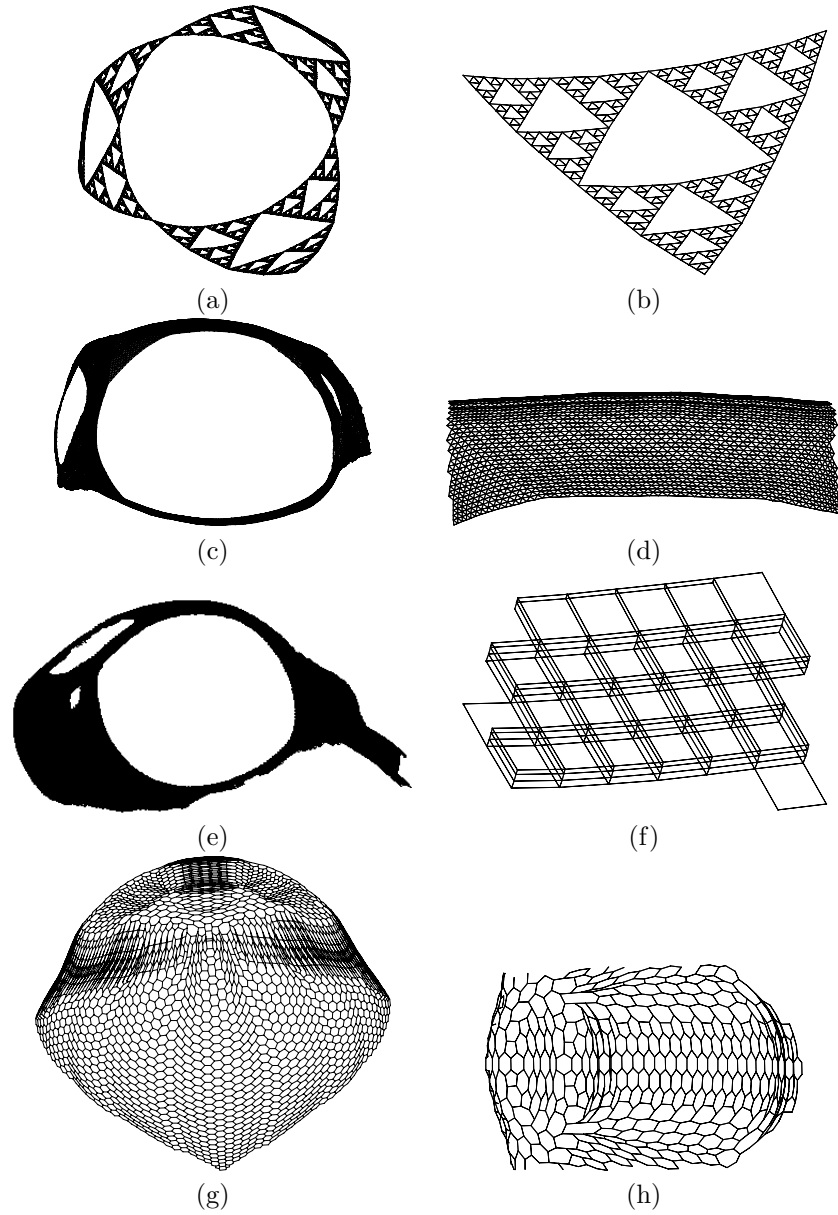


Figure 7: (a) A depth 7 Sierpinski graph; (b) zooming-in on the squeezed left side of (a); (c) the 4elt2 graph; (d) zooming-in on the bottom of (c); (e) the Ocean graph; (f) zooming-in on the micro structure of (e); (g) the 4970 graph; (h) zooming-in on the top-center portion of (g)

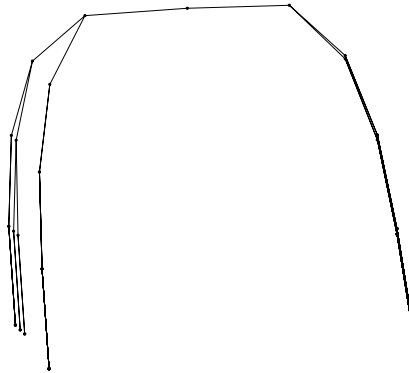


Figure 8: Drawing of a depth 5 full binary tree

ing able to draw a 10^5 -node graph in less than three seconds. Moreover, the implementation of our algorithm is much simpler and is almost parameter-free.

Recently, we have designed another algorithm for drawing huge graphs, which we call ACE [18]. ACE draws a graph by quickly calculating eigenvectors of the Laplacian matrix associated with it, using a special algebraic multigrid technique. ACE can draw 10^5 -node graphs in about 2 seconds. However, the running-time of ACE (like that of force-directed methods) depends on the graph’s structure, unlike our algorithm, where it depends only on the graph’s size. A detailed comparison between the results of the two algorithms has yet to be done.

In terms of drawing quality, the results of the new algorithm resemble those of force-directed graph drawing algorithms. However, being limited by the linear projection, frequently, the static 2-D results are inferior to those of the force-directed approach. For example, in many of the drawings that were given here, it may be observed that the boundaries are somewhat distorted, as they lie inside an absent third dimension. Nevertheless, we should stress the fact that the full power of our algorithm is not expressed well in static 2-D drawings. In order to really utilize its capabilities, one should explore the graph using the novel technique for interactive visualization, which is unique to this algorithm.

Several force-directed graph drawing algorithms compute the layout by first constructing a multidimensional drawing of the graph and then transforming it into a lower dimension, see, e.g., [9, 20]. Calculation of the initial multidimensional drawing is carried out by several kinds of force-directed optimization processes. Running time of such algorithms significantly increases as the dimensionality of the original drawing grows. Consequently, the dimensionality of the initial drawing is typically at most three or four. Regarding the transformation into a lower dimension, the work of [9] uses random projections, while the other method [20] uses a more computationally intensive optimization process that gradually decreases the “size” of one of the dimensions (a sort of “squashing”) until the drawing is of a lower dimensionality. Both these algorithms could pos-

sibly benefit from incorporating principal components analysis to project the drawings onto the plane. See also Subsection 6.1.

6 Additional Applications

6.1 Finding the best viewpoint for 3-D layouts

The popularity of 3-D graph drawing created a need for algorithms that automatically compute the best 2-D projection of a 3-D layout. An exact algorithm that requires a preprocessing time of $\Omega((|V|+|E|)^4 \log(|V|+|E|))$ was suggested in [6]. Iterative heuristic algorithms, with a faster running time (but still, at least $O(|V| \cdot |E|)$), were suggested in [13]. We suggest using PCA for calculating the best viewpoint, resulting in a much faster exact algorithm.

A good projection should prevent overlaps between the graph elements, avoiding vertex-vertex, vertex-edge, and edge-edge occlusions [6, 13]. Therefore, we construct a set of points, $\mathcal{P} \subset \mathbb{R}^3$, containing those points in the 3-D layout whose overlap we want to avoid in the 2-D projection. The set \mathcal{P} naturally contains all the 3-D points where the graph vertices are located. Sample points along which edges pass might also be added to \mathcal{P} .

Let us denote a projection by the function $p : \mathbb{R}^3 \rightarrow \mathbb{R}^2$. In a 2-D projection we want to minimize occlusions between points in \mathcal{P} . Hence, an adequate cost function for the quality of the projection will be:

$$\sum_{a \neq b \in \mathcal{P}} |p(a) - p(b)|^2. \quad (2)$$

A good projection should *maximize* (2), thus, separating elements of \mathcal{P} . In other words, the best projection is one that best preserves the pairwise squared distances of \mathcal{P} . The work of [17] proves that the maximizer is exactly the PCA projection of the points in \mathcal{P} . Therefore, we have an algorithm for finding the “best” viewpoint, whose running time is $O(|\mathcal{P}|)$. It is also possible to maximize a weighted version of (2), where we can grant more importance to preserving distances among some pairs. This is done by fixing non-negative pairwise weights and maximizing:

$$\sum_{a \neq b \in \mathcal{P}} w_{ab} |p(a) - p(b)|^2. \quad (3)$$

As explained in [17], such a maximization problem can be solved optimally and some general weighting schemes are suggested there.

We have not implemented this viewpoint-finding algorithm, so we cannot compare its quality to previous approaches. However, it constitutes a promising direction because of its excellent complexity and our positive experience with the aesthetical properties of PCA-based projection.

6.2 Applications to information visualization

Here, we consider a general scenario in which we are given some kind of data in a metric space, and we want to convey its overall structure by mapping it into a low-dimensional space (mostly 2-D or 3-D) that can be assessed by our own visual system. Using the pairwise distances between data elements, we can model the data by a weighted graph, and use force-directed drawing algorithms for computing a low-dimensional representation of the data. Our algorithm can deal directly with edge-weighted graphs, making it suitable for such an information visualization task.

In this case, our algorithm has an important performance-related advantage over other algorithms — fewer pairwise distances should be computed. Note that the time needed for computing the distance between two objects depends solely on the complexity of those objects, and is independent of n , the number of objects. (This is unlike the computation of the graph theoretic distance, which is not needed in this case.) Frequently, computing the distance between two objects is a costly operation; e.g., when the objects are DNA sequences of length k , a common distance measure is the “edit-distance”, whose computation may take time $O(k^2)$. Since in such applications n is typically large, one would have to consider multi-scale enhancements, and these would require the computation of the close neighbors of each of the objects. This, in turn, would require the computation of the distances between all pairs, resulting in $n \cdot (n - 1)/2$ distance computations, which is often far too costly. In contrast, our method requires only $m \cdot n$ distance computations — a significant improvement.

7 Conclusions

We have presented an extremely fast approach to graph drawing. It seems that our two key contributions are the simple technique for embedding the graph in a very high dimension and the use of principal component analysis (PCA) for finding good projections into lower dimensions.

The output of our algorithm is multi-dimensional, allowing multiple views of the graph. This also facilitates a novel technique for interactive exploration of the graph, by focusing on selected portions thereof, showing them in a way that is not possible in a 2-D drawing of the entire graph.

A newer work [16] suggests an alternative approach to the PCA projection. Thus, the second step of our algorithm is replaced by a new projection method that maximizes the scatter of the nodes while keeping edge lengths short.

References

- [1] I. Brass and A. Frick. Fast interactive 3-D graph visualization. In *Proc. 3rd Graph Drawing (GD'95)*, pages 99–110. Lecture Notes in Computer Science, Vol. 1027, Springer-Verlag, 1996.
- [2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
- [3] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. *ACM Trans. on Graphics*, 15:301–331, 1996.
- [4] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999.
- [5] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.
- [6] P. D. Eades, M. E. Houle, and R. Webber. Finding the best viewpoints for three-dimensional graph drawings. In *Proc. 5th Graph Drawing (GD'97)*, pages 87–98. Lecture Notes in Computer Science, Vol. 1353, Springer-Verlag, 1997.
- [7] B. S. Everitt and G. Dunn. *Applied Multivariate Data Analysis*. Arnold, 1991.
- [8] T. M. G. Fruchterman and E. Reingold. Graph drawing by force-directed placement. *Software-Practice and Experience*, 21:1129–1164, 1991.
- [9] P. Gajer, M. T. Goodrich, and S. G. Kobourov. A multi-dimensional approach to force-directed layouts of large graphs. In *Proc. 8th Graph Drawing (GD'00)*, pages 211–221. Lecture Notes in Computer Science, Vol. 1984, Springer-Verlag, 2000.
- [10] R. Hadany and D. Harel. A multi-scale method for drawing graphs nicely. *Discrete Applied Mathematics*, 113:3–21, 2001.
- [11] D. Harel and Y. Koren. A fast multi-scale method for drawing large graphs. *Journal of Graph Algorithms and Applications*, 6:179–202, 2002.
- [12] D. S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, 1996.
- [13] M. E. Houle and R. Webber. Approximation algorithms for finding best viewpoints. In *Proc. 6th Graph Drawing (GD'98)*, pages 210–223. Lecture Notes in Computer Science, Vol. 1547, Springer-Verlag, 1998.
- [14] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31:7–15, 1989.

- [15] M. Kaufmann and D. Wagner, editors. *Drawing Graphs: Methods and Models*. Lecture Notes in Computer Science, Vol. 2025, Springer-Verlag, 2001.
- [16] Y. Koren. Graph drawing by subspace optimization. In *Proceedings of 6th Joint Eurographics - IEEE TCVG Symp. Visualization (VisSym '04)*, pages 65–74. Eurographics, 2004.
- [17] Y. Koren and L. Carmel. Robust linear dimensionality reduction. *IEEE Transactions on Visualization and Computer Graphics*, 10:459–470, 2003.
- [18] Y. Koren, L. Carmel, and D. Harel. ACE: A fast multiscale eigenvector computation for drawing huge graphs. In *Proceedings of IEEE Information Visualization (InfoVis'02)*, pages 137–144. IEEE, 2002.
- [19] A. Quigley and P. Eades. FADE: Graph drawing, clustering, and visual abstraction. In *Proc. 8th Graph Drawing (GD'00)*, pages 183–196. Lecture Notes in Computer Science, Vol. 1984, Springer-Verlag, 2000.
- [20] D. Tunkelang. *A Numerical Optimization Approach to General Graph Drawing*. PhD thesis, Carnegie Mellon University, 1999.
- [21] C. Walshaw. A multilevel algorithm for force-directed graph drawing. In *Proc. 8th Graph Drawing (GD'00)*, pages 171–182. Lecture Notes in Computer Science, Vol. 1984, Springer-Verlag, 2000.
- [22] D. S. Watkins. *Fundamentals of Matrix Computations*. Wiley, 1991.