Consistent Graph Layout for Weighted Graphs by a Force-Based Probabilistic Algorithm

Dana Vrajitoru
Indiana University at South Bend
Computer and Information Sciences
http://www.cs.iusb.edu/~danav/
danav@cs.iusb.edu

TR-20040305-1

March 5, 2004

Abstract

In this paper we present three algorithms that build graph layouts for undirected, weighted graphs. Our goal is to generate layouts that are consistent with the weights in the graph. All of the algorithms are force-oriented and have been successful in solving the problem up to a certain precision. They all start with a random layout and improve it by iteratively repositioning the vertices to reduce the current error. The first two methods move the vertices along one edge at a time, either by selecting it randomly, or by following a breadth-first strategy. The third method computes the result of all of the tension forces occurring in each vertex and moves all of them in each step along the resulting vectors. We also show that if we start building the layout with a robust method and then refine the configuration with a more precise one, we can improve the quality of the solution.

1 Introduction

Let us suppose that millions of years from now aliens discover traces of human civilization on Earth and they attempt to recover our history from them. Moreover, suppose that the continents have derived from the form that they have today, and that all that the aliens can find is a schedule of an airline company featuring the duration of various flights from a location to another. The question is, can the aliens reconstruct the current map of the world based on that timetable?

To express this problem in mathematical terms, given an undirected and weighted graph, we must assign a 2D or 3D point to each of the vertices in the graph (a layout) such that for every two vertices A and B such that the edge A, B exists in the graph, the distance between the points assigned to them is equal to the weight of the edge.

Extensive work has been accomplished on drawing unweighted graphs with emphasis on showing the structure of the graph in the geometrical representation (Battista et al. [15], Diaz, Petit, and Serna [5]). Layouts presenting some aesthetic qualities are also appreciated (Gajer and Kobourov [12], Nesetril [16]). The problem is even more interesting and challenging when the graphs to be drawn are large (Gajer and Kobourov [12], Erlingsson and Krishnamoorthy [11], Brandes and Wagner [3], Hadany and Harel [13]). Another approach is to build the graph layout according to constraints that can be user-defined (Dornheim [7], Tamassia [18], He and Marriott [14]).

The best-known heuristic for generating graph layouts is certainly the spring algorithm (Eades [8]) that regards the edges in the graph as springs connecting the vertices such that the springs attract the vertices if they are too far apart and repel them if they are too close. In addition, non-connected vertices repel each other. In the usual implementation, the edges are expected to have the same length. An interesting model (Branke, Bucher, and Schmeck [4]) combines this method with the use of genetic algorithms to take into account other optimization criteria like the number of edge crossings or the number of different angles in the drawing.

Part of the research on graph layouts has also focused on weighted graphs and the best methods seems to be force-oriented (Battista et al. [15], Eades and Kelly [10], Bodlander et al. [1]). In one approach, Eades and De Mendonça [9] solve the triangulation conflicts in the graph by creating copies of certain vertices to obtain not only an equilibrium layout but also one which is completely tension-free.

Among the applications of these algorithms we can cite designing electronic circuits (Battista et al. [15]), designing web sites and visualizing the content of the World Wide Web (Brandes et al. [2]), parallel computing and VLSI (Diekmann et al. [6]).

The methods we present in this paper can be seen as variations of the spring algorithm (Eades [8]) in which we ignore the repulsion force exerted by non-adjacent vertices in the graph. The criteria we are interested in is the consistency between the distances between vertices in the graph and the weights of the edges.

The paper is structured the following way: Section 2 introduces the problem and two of our models that minimize the error in the graph. Section 3 discusses the existence of a solution and presents a third method that can find an equilibrium layout even when it is not possible to generate an exact solution. Section 4 presents some experimental results that validate our approach. Section 5 discusses possible graph topologies and the geometrical similarity between the original configuration of the graph and the layout found by our algorithms.

2 Layouts Minimizing the Error

In this section we introduce the problem and present two methods aiming to generate layouts minimizing the error in the graph defined as the absolute difference between the weights of the edges and the Euclidian distance between the vertices.

2.1 The Problem

Definition. Let $G = \{\mathcal{V}, \mathcal{E}\}$ be a graph where $\mathcal{V}$ is the set of vertices, $|\mathcal{V}| = n$, and $\mathcal{E}$ is the set of edges, $|\mathcal{E}| = m$. A *layout* for the graph is a function $P : \mathcal{V} \to \mathbf{R}^p$ that maps each vertex $v \in V$ to a geometrical point in $\mathbf{R}^p$, where usually $p = 2$ or 3. The edges are represented as line segments between the points associated with the vertices composing them.

For any two vertices $u, v \in \mathcal{E}$, we will denote the undirected edge $\{u, v\} \in \mathcal{V}$ by uv.

Problem. Let $G = \{\mathcal{V}, \mathcal{E}, W\}$ be an undirected, weighted graph where the weights of the edges are given by the function $W : \mathcal{E} \to \mathbf{R}^+$. We must find a layout $P : \mathcal{V} \to \mathbf{R}^3$ such that $\forall\, u, v \in \mathcal{V},\ \mathrm{d}\,(P(u), P(v)) = W(uv)$, the weight of the edge uv. A layout with this property will be called a *consistent layout* for this graph.

If $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$, then we must find a set of points $\{P_1, P_2, \ldots, P_n\}$ such that if there is an edge between two vertices v_i and v_j, $v_i v_j \in \mathcal{E}$, then the points associated with these vertices are placed at a distance from each other equal to the weight of the edge.

$$\mathrm{d}\,(P_i, P_j) = W(v_i v_j) \tag{1}$$

We can express the constraints in Equation 1 as a system of m equations of second degree with $3n$ variables. Let us denote each of the points as a 3-dimensional vector $P_i = (x_i, y_i, z_i), 1 \leq i \leq n$, and the weight of the edge $v_i v_j \in \mathcal{E}$ by w_{ij}. Then for each edge $v_i v_j \in \mathcal{E}$, we have the following equation:

$$(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2 = w_{ij} \tag{2}$$

This system of equations has either no solution, or an infinity of them. Any isometric geometrical transformation, for example, a translation, rotation, or symmetry, applied to a consistent layout, transforms it into another consistent one.

This problem has been proven to be NP-hard (Eades and Mendonca [9]). Moreover, the entailed equations can be stiff, which makes them hard to solve.

2.2 Our Model

Following the ideas from the the spring algorithm and most of the force-oriented methods, the graph forms a dynamic system in which each element (vertex) is attracted or repelled by its neighbors according to the discrepancy between the distance between the points assigned to the vertices and the weight of the edge they form in the graph. In our model, if two vertices are not neighbors in the graph, then they do not interact directly with each other.

The algorithms to be presented in this section consist in passing from one state of the system to another one of greater probability. Both of them reposition one vertex at a time in such a way as to reduce the error on one of the edges starting from it.

The first algorithm, that we refer to as the *random edge (RE)* algorithm, chooses an arbitrary edge in the graph at each iteration and moves one of the points associated with the vertices composing the edge. The point is moved on the line containing the two points, further away from the second point if the distance is smaller that the weight of the edge, and closer to the second point if the distance between them is greater than the weight of the edge.

Let A and B be two vertices in the graph such that the edge AB is present in the graph with the weight W_{AB}. Suppose that the random function has chosen this edge at a particular iteration of the algorithm and that $P_A = (x_A, y_A, z_A)$ and $P_B = (x_B, y_B, z_B)$ are the points currently assigned to the vertices. Let us denote by err_{AB} the error on the edge AB computed as the difference between the weight of the edge and the Euclidian distance between the two points:

$$err_{AB} = W_{AB} - d(P_A, P_B). \tag{3}$$

This error gives us an estimation of how much the points are misplaced with respect to each other considering that the weight of the edge represents the ideal distance between them. If the error is positive, then the points are too close to each other. If the error is negative, the points are too far apart.

If the error is not equal to 0, we will adjust the position of the vertex B by assigning it a new point P'_B determined in the following way:

$$P'_B = P_B + \varepsilon \cdot \frac{err_{AB}}{d(P_A, P_B)} \cdot (P_B - P_A), \tag{4}$$

where ε is a constant, $0 < \varepsilon < 1$.

In this formula, if the error is positive, then the point P_B will be moved away from P_A on the line containing P_A and P_B . If the error is negative, the point P_B will be moved closer to P_A on the same line.

To justify the above formula, let us notice first that the new length of the edge is closer to the weight of the edge than the previous one. Thus, we can calculate:

$$d(P_A, P'_B) = \left| \varepsilon \cdot \frac{err_{AB}}{d(P_A, P_B)} + 1 \right| \cdot d(P_A, P_B) = \varepsilon \cdot err_{AB} + d(P_A, P_B)$$

The new error associated with the edge (A, B) is

$$\begin{aligned} err'_{AB} &= W_{AB} - d(P_A, P'_B) = (W_{AB} - d(P_A, P_B))(1 - \varepsilon) \\ &= err_{AB}(1 - \varepsilon) \end{aligned}$$

Since we know that $0 < \varepsilon < 1$, we can conclude that

$$|err'_{AB}| < |err_{AB}|$$

Thus, the procedure reduces the error on this particular edge. Moreover, we can note two things. First, if $\varepsilon = 1$, then the new error will be null: $err'_{AB} = 0$. Second, if we iterate the modification of P_B that we have described, the error is converging to 0 because we multiply it at each iteration with a positive constant that is less than 1.

The algorithm will most probably not choose the same edge for the next iteration, but another arbitrary edge in the graph. The previous observations do not guarantee that the error on every edge will be converging to 0 because the improvement of the error on one edge can deteriorate the error on another one. The experimental results to be presented in Section 4 have shown that the total error in the graph decreases in general if we compute it periodically after a number of iterations equal to a multiple of the number of edges. Still, iteration by iteration, this is not a completely monotone trend.

The parameter ε allows us to control the amount of adjustment that is performed at each step and thus, decide on the convergence rate.

Here is the pseudocode version of the algorithm that we have just described:

```
for a number of iterations
   for the number of edges m
      select_random_edge(A, B);
      adjust_edge(point[A], point[B], weight[A, B], epsilon);
```

The second algorithm that we refer to as the *breadth-first scan (BFS)* algorithm we propose uses the same method to adjust an edge (Equation 4), but it does not choose the edge randomly. At each iteration, the algorithm starts with a randomly chosen vertex (origin), and it adjust all the other vertices in the graph starting from this origin with a breadth-first scanning method. Thus, the direct neighbors of the origin will be adjusted in the first steps, then all of their neighbors, and so on. The adjustment is spreading in the graph as a wave starting from the origin. The only random component in this variant is the choice of the origin.

This method presents the advantage that when a vertex is moved on an edge, we know that by decreasing the error on that edge, we don't affect the edges considered beforehand in that iteration, only edges to be visited afterward or not at all. Thus, we expect this algorithm to decrease the total error more consistently than the first one, which is actually what we have observed in our experiments (Section 4). By starting from a different origin at every iteration, we insure that the layout will not prematurely converge to a suboptimal configuration and that all of the edges in the graph will be visited at some point.

The second algorithm has the following pseudocode:

```
for a number of iterations
   queue = empty;
```

```
origin = random(number_of_vertices);
queue += origin;
while (queue is not empty)
   A = queue--;
   for every B, a neighbor of A
      if (B has not been in queue)
         adjust_edge(point[A], point[B],
                     weight[A, B], epsilon);
         queue += B;
```

There are two distinct sets of problems that we must consider for testing and validating our algorithms.

In the first category, we have graphs for which there exists a consistent layout. For this category, we expect the breadth-first scan algorithm to converge faster to a possible solution.

In the second category, we have problems for which there is no consistent layout. In this case, we aim to find the layout that is closest to an equilibrium, or in other words, that minimizes the total edge error in the graph. We think that in some cases the random edge algorithm could find better solutions for this category of problems.

3 Approximate Solutions

The algorithms that we have presented work well when there exists a solution to the problem. In this section, we would like to express the requirements for an approximate optimal solution in the case where an exact solution does not exist and describe a third algorithm that is more appropriate for this case.

3.1 Minimal Total Error

The necessary conditions for the existence of an exact solution are related to the properties of the Euclidian distance. Thus, if the weights of the edges represent actual distances, then they must satisfy the following conditions:

$$\forall\, A,\, B\, \in \mathcal{V}, \quad W_{AB} \geq 0 \tag{5}$$

$$\forall\, A,\, B\, \in \mathcal{V}, \quad W_{AB} = W_{BA} \tag{6}$$

$$\forall\, A,\, B,\, C\, \in \mathcal{V}, \quad W_{AC} \leq W_{AB} + W_{BC} \tag{7}$$

For an undirected graph, the conditions 5 and 6 are trivial. Equation 7 is also known as the triangulation condition.

These constraints represent necessary but not sufficient conditions for the existence of the solution. For example, the following graph satisfies all of these conditions, but no layout for this graph can be consistent with the weights.

The constraints expressed in Equations 5, 6, and 7 are not sufficient conditions for the existence of the solution even in the case of a completely connected

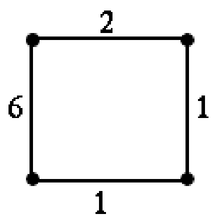

Figure 1: A graph with no solution

graph. For example, the following graph satisfies all of the conditions and is completely connected, but no layout can be consistent with its weights. It can be proven that given the weights in the triangles ΔABC and ΔABD, the minimal and maximal distances between the vertices C and D must be approximately 3.623 and 10.764 respectively. The minimal distance between C and D is attained when the two triangles are coplanar and C and D are on the same side of the line AB. The maximal distance is attained when the triangles are coplanar but on opposed sides of the line AB.

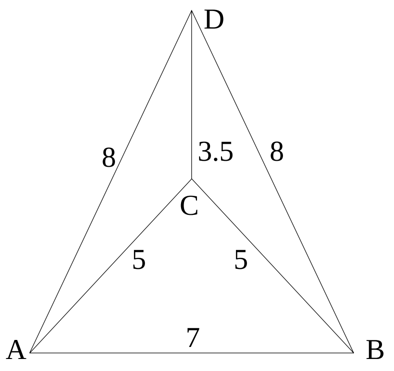

Figure 2: A complete graph with no solution

To verify the condition 7 for any three vertices in the graph, we must dispose of the edges forming the triangle, and this is not always the case. It seems appropriate to extend the triangulation condition to any polygon or cycle in the graph, as expressed in Equation 8. Again, as before, this is a necessary but not sufficient condition for the existence of the solution.

$$\begin{gathered}\forall n \in \mathbf{N},\ n \geq 3, \quad \forall A_1,\ A_2,\ \ldots,\ A_n \in \mathcal{V}, \\ W_{A_1 A_n} \leq W_{A_1 A_2} + W_{A_2 A_3} + \ldots W_{A_{n-1} A_n}\end{gathered} \tag{8}$$

Although an algorithm that verifies the condition 8 would be exponential, it is much easier to generate graphs for which we know that there is a solution. For this, we can start with an unweighted graph, assign 3D points to the vertices, and then set the weights according to the Euclidian distance between these points.

The same way, it is easy to generate graphs for which the problem is insolvable. For this, the graph must contain at least one cycle, because there is always a solution for a tree. We can set the weights in a cycle of the graph such that the constraint 8 is not satisfied. This operation is linear in the selected cycle.

In the case where there is no solution for a given graph, we would like to find a layout that minimizes the total absolute error in the graph:

$$total\_error = \sum_{\forall AB \in \mathcal{E}} |err_{AB}| \tag{9}$$

The algorithms presented in the previous section are designed to minimize the total error. In the next paragraph we introduce a third algorithm that aims to find an equilibrium configuration in which the tension generated by the error on all the edges connected to each vertex compensate each other.

3.2 Equilibrium Layout

Let us suppose that we can construct a physical representation of the graph using interconnecting springs for the edges, as in the spring method. Each spring corresponding to an edge would have an initial length equal to the weight of the edge and a section much smaller than the length. These springs can only be deformed along the main direction. When extended, the springs tend to contract to their initial length, and when compressed, they tend to extend. Moreover, each spring creates a contracting or extending force along the main direction proportionate to the amount of deformation that was applied to it.

We can build the graph using these springs by deforming them as necessary to fit the connections in the description of the graph. The physical construction would then naturally evolve to an equilibrium state in which the deformation tensions compensate each other, if they are not solved.

With the next algorithm we try to find the equilibrium solution for the situation that we have described.

We focus again on the points representing the vertices in the graph. For each point, a number of forces exert on it as a result of the deformation along the edges connected to the vertex. If the result of all the forces is not 0, then the point will be pushed in the direction of the resulting force.

We can now express the condition for the solution with no local tension. We would like to find a layout for the graph such that the result of all the deformation forces that exert on each vertex is a null vector. An approximate solution must minimize the total norm of the resulting tension force in each vertex. We believe that for any graph, there exists at least one equilibrium solution.

In the following algorithm, that we will refer to as the *tension vector (TV)* algorithm, for each of the edges that has suffered deformation, opposite forces of equal norm are exerted on the vertices composing the edge. Thus, the result of all the tension forces in the graph is always 0.

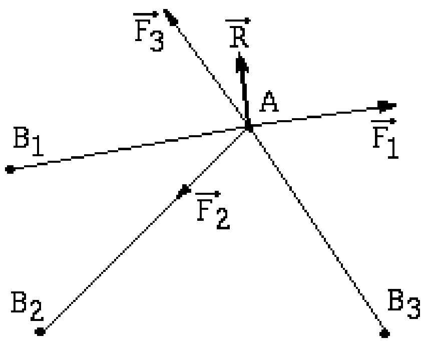

Figure 3: Resulting tension force

For example, let us suppose that a vertex A is connected to three vertices B_1, B_2, B_3 as in Figure 3.

On each of the edges $AB_i \in \mathcal{E}$, $i = 1, 2, 3$, the deformation suffered by the edge engenders a force proportional to it in the contrary direction, that we have denoted by $\vec{F}_i$, $i = 1, 2, 3$. Thus, from the direction of these forces we can deduce that the points corresponding to the vertices B_1 and B_3 are closer to the point associated with the vertex A than they should be. On the contrary, the point associated with B_2 is farther from the point assigned to A than indicated by the weight of that edge.

By composing the three deformation forces $\vec{F}_i$, $i = 1, 2, 3$, we obtain the resulting force that applies to A, denoted by $\vec{R} = \vec{F}_1 + \vec{F}_2 + \vec{F}_3$. The algorithm assumes that the point corresponding to A will be moved along $\vec{R}$ until the resulting force is null.

We still have to define the the deformation force in a precise way. We can start by the amount of deformation err_{AB} which has been defined in Equation 3 as the difference between the weight of the edge AB and the distance between the points associated with the two vertices, P_A and P_B. Then we can define the deformation force applied to the point P_B as being

$$\vec{F}_{AB} = err_{AB} \frac{\vec{AB}}{\|\vec{AB}\|} \tag{10}$$

Thus, is the error is positive, then the two points are too close and P_B should move away from P_A, which is in the direction of the vector $\vec{AB}$.

In Equation 10, we have assumed that the deformation suffered by the edge AB is equally distributed between the two points. Thus, for an undirected graph, for each force $\vec{F}_{AB}$, there is a corresponding opposing force equal in norm applied to the other extremity of the edge:

$$\vec{F}_{AB} = -\vec{F}_{BA}$$

We could extend the model to oriented graphs by computing the tension vector on an edge as the average between the tension vectors resulting from applying Equation 10 to each direction of the edge. If the edge goes in only one direction, the tension vector on the other direction would be 0. For example, the force applied to P_B as a result of the vertex A is equal to:

$$\vec{F}_{(A)B} = \frac{1}{2}(\vec{F}_{AB} - \vec{F}_{BA})$$

Then we can define the resulting force applied to the point P_A:

$$\vec{R}_A = \sum_{\forall AB \in \mathcal{E}} \vec{F}_{BA} \tag{11}$$

If P_A is the point associated with the vertex A in a particular iteration and $\vec{R}_A$ is the force exerted on it, the algorithm moves the point to a new location P'_A defined as follows:

$$P'_A = P_A + \varepsilon \vec{R}_A \tag{12}$$

where ε is a constant, $0 < \varepsilon \leq 1$.

At last, the algorithm starts again with a random layout and moves the points according to Equation 12 in a given number of iterations or until the layout convergences to an equilibrium. In each iteration, all of the tension forces are computed in the first step, then all of the points are moved in the next step without recomputing the forces. The following is a general description of the algorithm in pseudocode:

```
for a number of iterations
   for all A in V
      compute RA;
   for all A in V
      PA = PA + epsilon * RA;
```

In this algorithm, the tension force in every vertex is computed based on the current layout before any of them is moved. This is the major difference between this algorithm and the previous ones introduced in Section 2.2, that move one point at a time and reevaluate the situation after each of them.

4 Experimental Results

We have conducted our experiences with two sets of problems, the first one containing 10 weighted graphs for which there is at least one known solution to the problem, and the second one containing 10 graphs for which an exact solution probably doesn't exist.

In the first set, the graphs have been generated in 3 steps:

- the unweighted graph has been generated by random;
- we have generated a random bounded 3D layout for the graph;

Table 1: Average results in 1000 iterations for graphs with existing solution

Graph	Total Error				Total TV Norm
	Initial	BFS	RE	TV	
dg30	9315.06	182.603	290.914	212.392	0.672
dg40	16406.87	60.014	230.970	68.650	0.231
dg50	26833.27	278.409	389.686	256.735	0.001
dg60	49896.58	1249.940	895.733	986.746	0.000
dg70	99540.28	780.534	998.758	208.130	3.259
dg80	117435.83	4.393	3.501	0.125	0.004
dg90	133947.17	6.330	1807.242	677.908	0.049
dg100	197523.57	3437.070	4466.200	442.295	0.449
dg125	289228.47	1732.634	2935.625	95.737	1.701
dg150	389771.23	34.621	4.835	0.406	0.009
dg175	475668.2	47.941	8077.074	0.470	0.009
dg200	670394.55	7.224	0.217	0.605	0.009

- the weights in the graph have been computed as the distance between the points assigned to vertices composing each edge.

For the second set, both the unweighted graphs and the weights have been generated randomly. From the results of the various trials on these graphs we can deduce that by generating the weights this way we have introduced several conflicts with Equation 8. Thus, there is no exact solution for these problems.

The results from the first set of problems are encouraging. All of the methods have converged to a solution very close to an exact one within a number of iterations depending on the size of the graph and on the number of connections. A higher number of connections can increase the speed of the convergence.

Table 1 shows the results of the three methods on the first set of problems. The graphs are named after their number of vertices. The numbers represent the total error in the graph after 1000 iterations as an average over 10 different trials with different seeds for the pseudo random number generator. The ϵ parameter is equal to 0.005 for these results. Higher values have caused the tension vector algorithm to diverge.

The last column in the table shows the total sum of the norms of the tension vectors in each vertex of the graph. We have included this column in the table because it illustrates that the algorithm in this case may converge to an equilibrium solution that does not minimize the total error in the graph. It would represent a physically unstable equilibrium. In these cases, the system has been stabilized, but any small perturbation in the position of one of the vertices could result in the system becoming unstable and converging to a different solution.

Table 2 shows the total error from Table 1 as a percentage of the sum of all of the weights in the graph. From this table we can notice that the error is less that 3% in all of the cases, and at least half of the time less than 1%. This

Table 2: Total error in 1000 iterations as percentage of the total weight in the graph, existing solution

Graph	BFS	RE	TV
dg30	1.48%	2.36%	1.72%
dg40	0.36%	1.39%	0.41%
dg50	0.80%	1.12%	0.74%
dg60	2.06%	1.48%	1.63%
dg70	0.64%	0.82%	0.17%
dg80	0.00%	0.00%	0.00%
dg90	0.00%	1.02%	0.38%
dg100	1.40%	1.82%	0.18%
dg125	0.46%	0.77%	0.03%
dg150	0.01%	0.00%	0.00%
dg175	0.01%	1.26%	0.00%
dg200	0.00%	0.00%	0.00%

means that all of the algorithms have found a solution that is more than 97% precise.

We can also remark from this table that the tension vector algorithm is in general more precise than the other two methods. The situations in which this is not the case are most probably due to the convergence of the graph to an unstable equilibrium solution. The criteria for deducing this is the fact that the sum of the norms of all the tension vectors in the graph is very small in these situations.

Table 3 shows the results of the three methods on the set of problems with no solution. The last column has the same meaning as before, showing that even if the solution found by the tension vector algorithm is far from being exact, it still represents an equilibrium point for the system.

Table 4 shows the total error in Table 3 as the percentage of the sum of all of the weights in the graph.

From this second set of results we can see that although there is no solution to these problems, all of the methods have found a graph layout that is much closer to the constraints than the original one. The third method also generates more precise solutions than the other for these problems, and the difference is even more visible than for the other set of problems. We can also notice from the last column in Table 3 that the tension vector method has generated layouts that are quite close to an unstable equilibrium solution.

To illustrate the behavior of the algorithms, we have plotted the average total error as it evolves through the first 500 iterations. Figures 4, 5, and 6 show these charts for the set of problems with existing solution, for the breadth-first scan, random edge, and tension vector algorithms respectively. Figures 4, 5, and 6 show the same charts for the set of problems with non-existing solution.

Table 3: Average results in 1000 iterations for graphs with non-existing solution

Graph	Total Error				Total TV Norm
	Initial	BFS	RE	TV	
ukn50	8177.43	2171.060	2278.012	2082.645	0.106
ukn60	13326.01	3988.101	4115.643	3871.154	0.308
ukn70	27483.10	9714.540	9640.596	9477.518	8.149
ukn80	32838.28	11498.750	11547.960	11282.030	8.549
ukn90	38774.58	13357.580	13412.910	13106.140	7.474
ukn100	53859.34	20306.400	20338.950	19842.180	8.221
ukn125	82358.92	31329.290	31348.580	30742.930	8.768
ukn150	110082.90	42273.580	42115.450	41624.430	8.659
ukn175	136130.17	54228.630	54227.700	53380.870	9.513
ukn200	191138.27	77473.520	77285.360	76503.800	12.191

Table 4: Total error in 1000 iterations as percentage of the total weight in the graph, non-existing solution

Graph	BFS	RE	TV
ukn50	27.27%	28.62%	26.16%
ukn60	29.98%	30.94%	29.10%
ukn70	37.45%	37.17%	36.54%
ukn80	36.75%	36.91%	36.06%
ukn90	37.19%	37.35%	36.49%
ukn100	38.79%	38.85%	37.90%
ukn125	40.29%	40.31%	39.53%
ukn150	41.16%	41.01%	40.53%
ukn175	41.52%	41.52%	40.87%
ukn200	42.19%	42.08%	41.66%

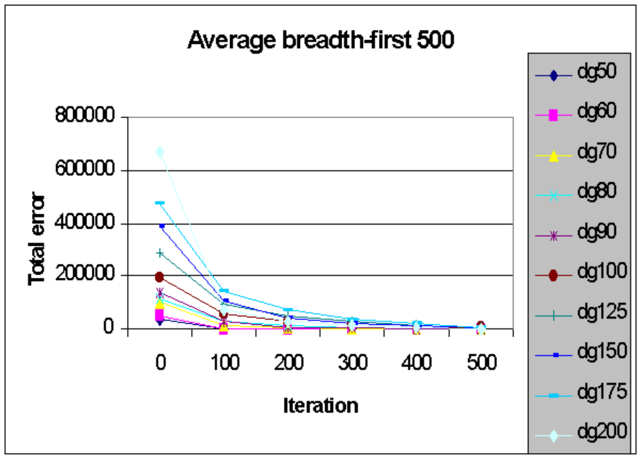

Figure 4: Average total error for BFS in 500 iterations, existing solution

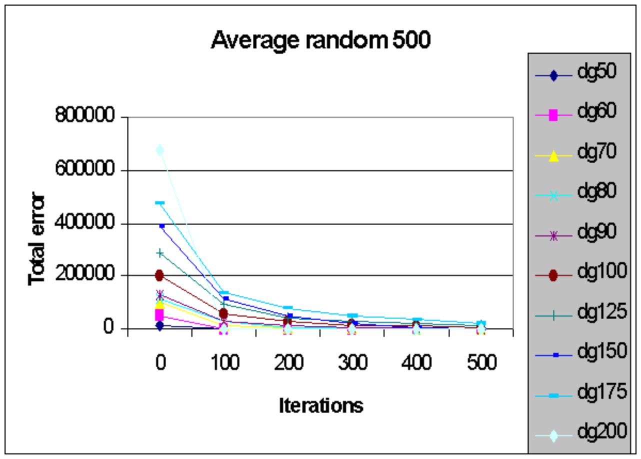

Figure 5: Average total error for RE in 500 iterations, existing solution

From these figures we can notice that the total error decreases very fast in the first few iterations, and then it's evolution is much slower for both sets of problems. To illustrate this phenomenon, Figure 10 shows the average total error for the second set of problems for the tension vector method in 200 iterations. This figure shows that in the 20 first iterations, the total error is adjusted a lot more than in the 180 next iterations for all of the problems.

Finally, Figure 11 shows the evolution of a graph with 125 vertices and 3000 edges through 1000 iterations under the tension vector algorithm and the layout of the graph at various stages of the computation. The edges are color coded with the following meaning: red for edges that are too long (the distance

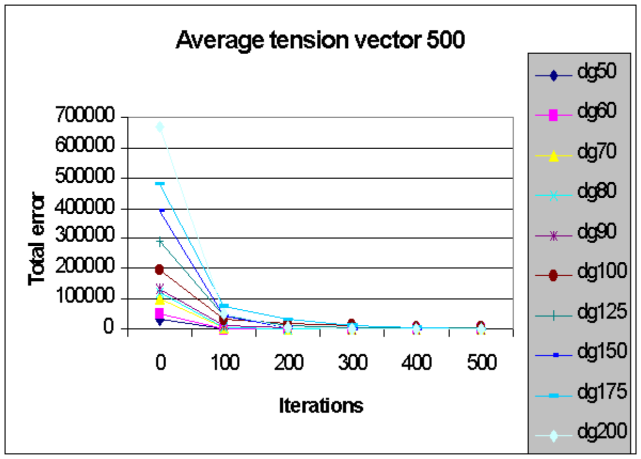

Figure 6: Average total error for TV in 500 iterations, existing solution

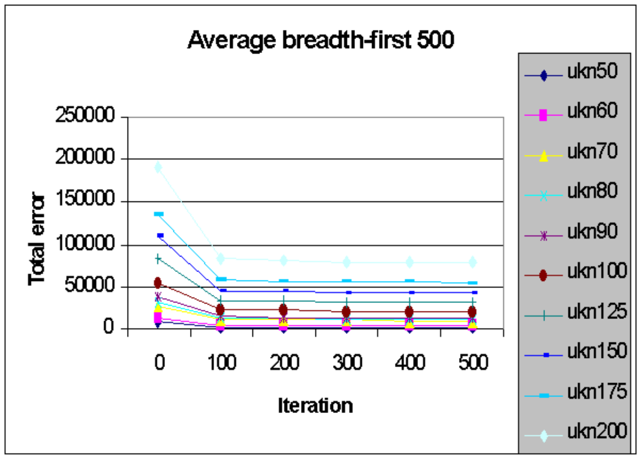

Figure 7: Average total error for BFS in 500 iterations, non-existing solution

between the points is greater than the weight of the edge), blue for edges that are too short, and yellow for edges of the right length. The images have been created in OpenGL using the DataViewer package [17].

4.1 Combining Methods

The previous results have shown that the tension vector algorithm is the one generating the most consistent layouts. Still, this method has a major disadvantage which is that for relatively large graphs (with more than 50 vertices), the algorithm diverges for values of the parameter ε that are not small enough. In

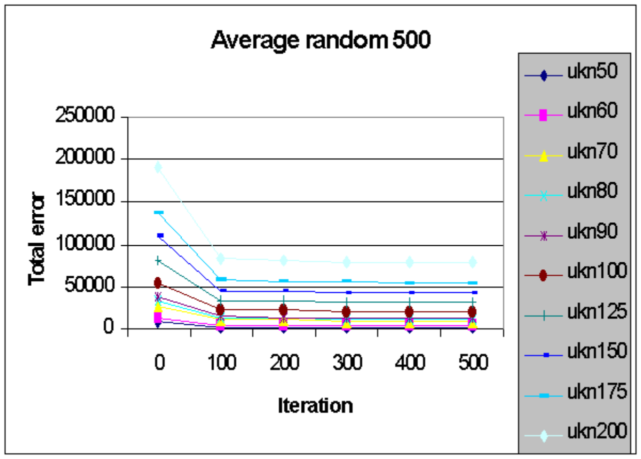

Figure 8: Average total error for RE in 500 iterations, non-existing solution

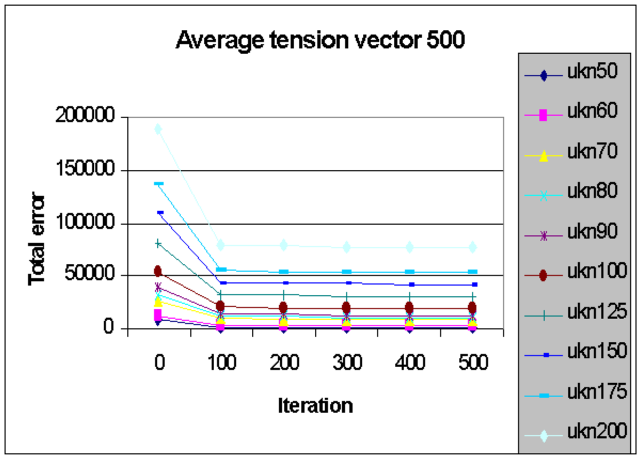

Figure 9: Average total error for TV in 500 iterations, non-existing solution

our example, the maximal value of ε that we could use was 0.005. This means that although the algorithm can build quite precise layouts, the limitation on the value of ε imposes a longer execution time to achieve to a certain degree of precision.

The breadth-first scan method on the other hand has never presented divergence problems, which means that we can use any value for ε. Higher values of the parameter lead to faster convergence of the layout to a given precision.

The last idea that present in this paper is to combine the two algorithms to take advantage of the strong points for each of them. We have performed a new set of experiments using 2 graphs with existent solution and 2 graphs with non-

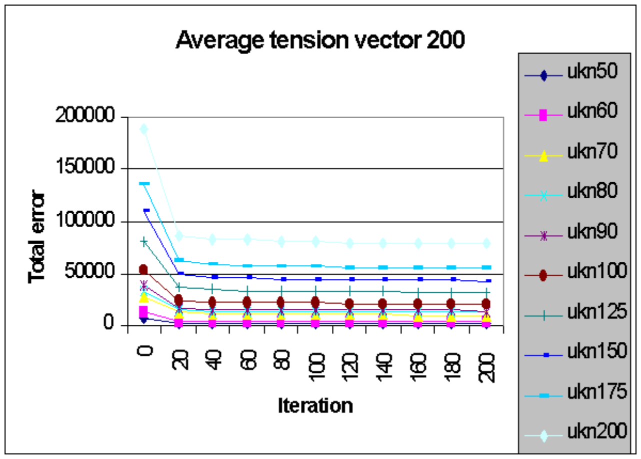

Figure 10: Average total error for TV in 200 iterations

Table 5: Average results with the combined method in 1000 iterations

Graph	BFS	TV	Combined
dg100	634.69	442.30	213.39
dg200	0.46	0.6	0.38
ukn100	19890.52	19842.18	19614.77
ukn200	76758.73	76503.80	75779.31

existent solution, with 100 and 200 vertices respectively. We start by applying the breadth-first scan method for 900 iterations with $\varepsilon = 0.05$, then we continue with the tension vector method for another 100 iterations with $\varepsilon = 0.005$.

Table 5 compares the results of this last method with the breadth-first scan algorithm on 1000 iterations with $\varepsilon = 0.05$, and with the tension vector algorithm also on 1000 iterations with $\varepsilon = 0.005$. From these results we can see that for the same amount of computation time, we can generate more precise layouts by combining the two methods. This also means that to attain a given precision, the combination of the two algorithms can work faster.

5 Topologies

We have seen that when the graph has a consistent layout, all of the algorithms can find approximate solutions with good precision. The next question we can ask is, when we start with a particular known layout for a graph, and we set the weights to be consistent with the Euclidian distance between the vertices in this layout, what is the chance of finding this exact configuration by any of

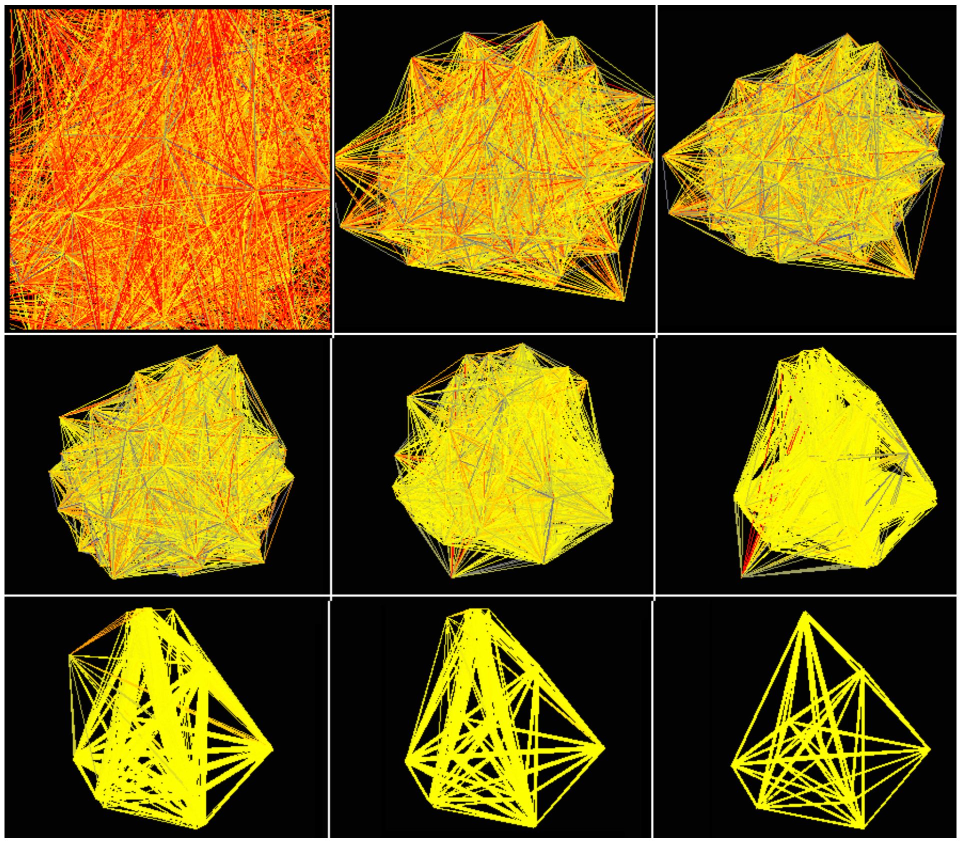

Figure 11: Evolution of a graph layout in 1000 iterations with TV

these algorithms?

An important observation is that if we find a solution to the problem, then we can also find an infinity of them by applying any isometric geometrical transformation to the original solution. Thus, the question becomes, what constraints can we impose on the structure of the graph and on the initial layout such that the solution we obtain is equivalent to the original solution modulo an isometric transformation?

We have experimented with several possible topologies for the original graph layout and compared them from a strictly visual point of view to the layouts generated by our algorithms.

Figure 12 shows on the left side a graph with 150 vertices and 1200 edges that is originally organized in a spiral. The image on the right side represents a solution found by the tension vector algorithm in 10000 iterations with an epsilon of 0.0005. From this figure we can notice a visual resemblance between the two figures, although they are probably not quite equivalent (by an isometry).

Figure 13 shows on the left side a graph with 100 vertices and 2000 edges

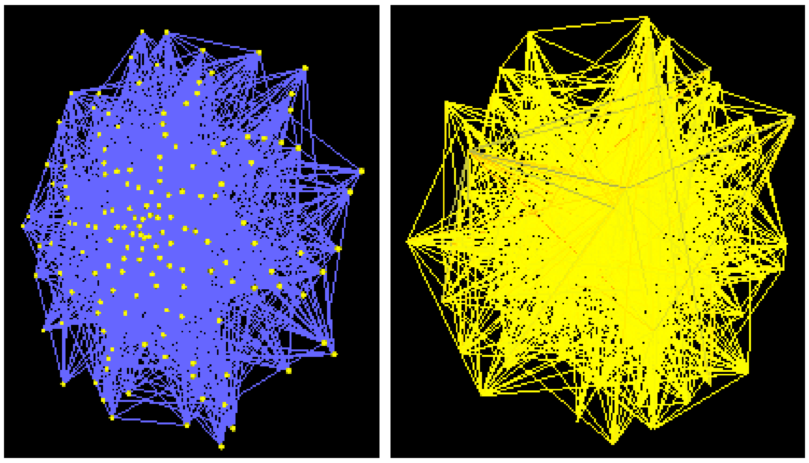

Figure 12: Original spiral layout and the solution found by TV

that is originally organized in a clustered layout. The vertices in each cluster are much closer to each other than they are to other clusters. What appears as one vertex in the figure is actually a whole cluster. The image on the right hand side represents a solution found by the tension vector algorithm in 10000 iterations with an epsilon of 0.0005. We can notice that the algorithm has reconstructed the cluster organization of the graph and the result is quite similar to the original one.

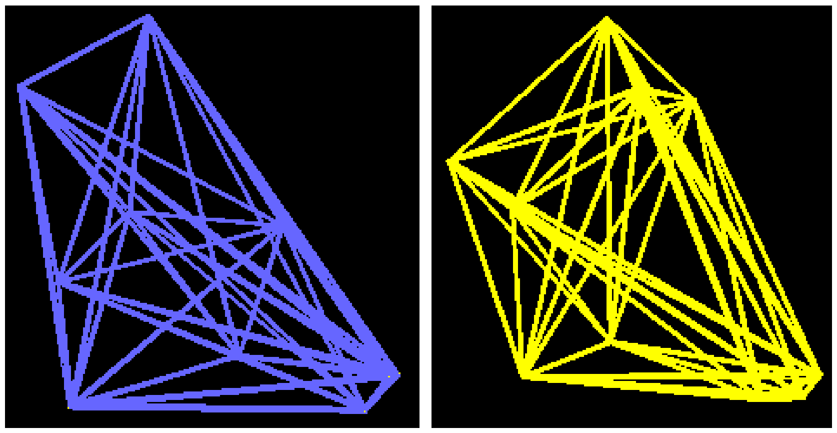

Figure 13: Original cluster layout and the solution found by TV

In the next two examples we can see two graphs for which the solution found by our algorithm is clearly very different from the original one. Figure 14 shows a graph with 227 vertices and 471 edges for which the original layout is on a sphere. Figure 15 shows a graph with 200 vertices and 400 edges, and with an original layout on a torus. The images on the right show solutions found by the tension vector algorithm in 10000 iterations with an epsilon of 0.0005. We can see that in both cases, the solution found by our algorithm looks nothing like the original layout of the graph.

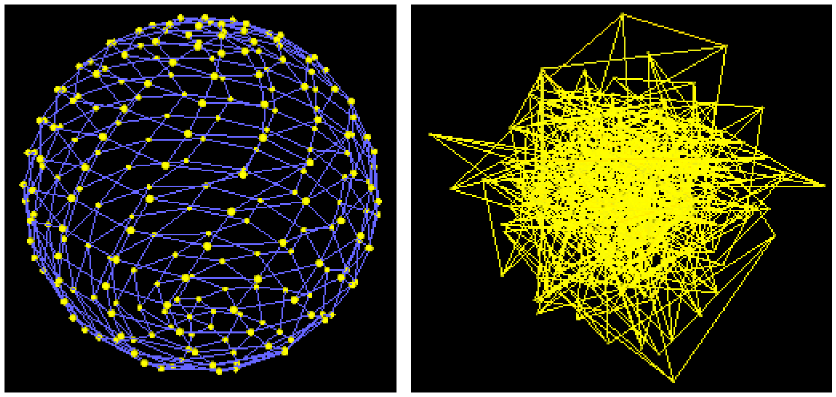
Figure 14: Original sphere layout and the solution found by TV

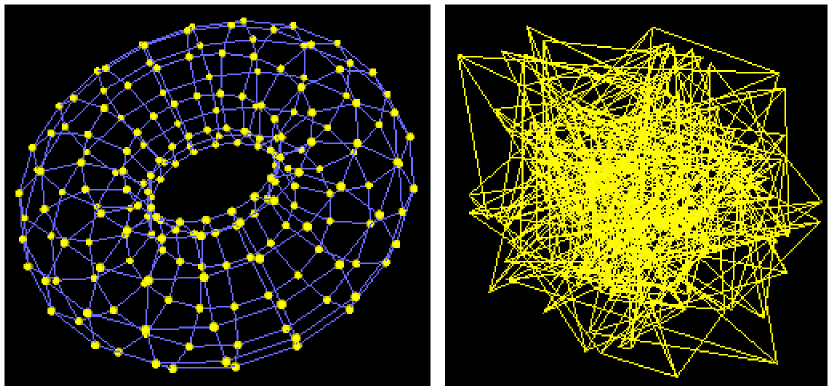
Figure 15: Original torus layout and the solution found by TV

One of the factors that could influence the resemblance between the original layout and the iterated solution is the number of edges in the graph. A higher number of edges for the same number of vertices imposes more geometrical constraints that restrict the number of non-equivalent solutions.

A second factor of influence is the flexibility of the original layout. In the case of the sphere and torus graphs, the vertices are in general linked to a limited number of the their close neighbors. As a result, we could deform the graph in many ways without modifying the distance between the vertices. This implies that there exist a large number of non-equivalent solutions.

Many of the existing force-based methods for building graph layouts, including the spring method, are taking into consideration repulsion forces between non-adjacent vertices in the graph. Part of our future research will be to incorporate this component in our methods as a constraint for building consistent graph layouts maximizing the enclosed volume. This could result in finding the original layout in the cases of the ellipsoid and torus topologies since the original layout is on a geometrical figure with this property.

6 Conclusion

In this paper we have presented three algorithms that aim to build graph layouts that are consistent with the weights in an undirected graph. All of the algorithms start with a random layout that they improve by iteratively decreasing the amount of error on the edges. All of them are based on the idea of attraction and repulsion forces between the vertices based on the Euclidian distance between the points and the weights. This idea is similar to the spring algorithm and other force-oriented methods.

The first two algorithms, breadth-first scan and random edge, modify one vertex at a time based on the information from one edge the vertex belongs to. They are robust methods that can be applied with a large range of choices for the parameters. The third method, named the tension vector algorithm, considers all of the edges associated with each vertex and moves all of the points in one step before recalculating all of the tension forces.

The experimental results have shown that all of the methods can generate consistent layouts with a precision of over 97% if the problem is solvable. In the case where it is not possible to generate a consistent layout, the algorithms can build configurations minimizing the total error or even find equilibrium solutions in the case of the tension-vector algorithm.

The best results clearly belong to the tension vector algorithm, although the phenomenon of divergence occurring in some cases makes the other methods valid alternatives to it. Finally, we have shown that combining the strength of several algorithms we can generate more precise layouts faster.

References

[1] H. Bodlaender, M. Fellows, and D. Thilikos. Derivation of algorithms for cutwidth and related graph layout problems. Technical Report UU-CS-2002-032, Institute for Information and Computing Sciences, Utrecht University, 2002.

[2] U. Brandes, V. Kääb, A. Löh, and D. Wagner. Dynamic WWW structures in 3d. *Journal of Graph Algorithms and Applications*, 4(3):183–191, 2000.

[3] U. Brandes and D. Wagner. Using graph layout to visualize train interconnection data. *Journal of Graph Algorithms and Applications*, 4(3):35–155, 2000.

[4] J. Branke, F. Bucher, and H. Schmeck. Using genetic algorithms for drawing undirected graphs. In J. Allen, editor, *The Third Nordic Workshop on Genetic Algorithms and their Applications*, pages 193–205, 1997.

[5] J. Daz, J. Petit, and M. Serna. A survey on graph layout problems. *ACM Computing Surveys*, 34(3):313–356, 2002.

[6] R. Diekmann, R. Lüling, and B. Monien. Communication throughput of interconnection networks. *Lecture Notes in Computer Science*, 841:72–86, 1994.

[7] C. Dornheim. Planar graphs with topological constraints. *Journal of Graph Algorithms and Applications*, 6(1):27–66, 2002.

[8] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[9] P. Eades and X. de Mendonça. Vertex splitting and tension-free layout. In *Graph Drawing*, number 1027 in Lecture Notes in Computer Science,, pages 244–253, 1995.

[10] P. Eades and D. Kelly. Heuristics for reducing crossings in 2-layered networks. *Ars Combin.*, 21.A:89–98, 1986.

[11] U. Erlingsson and M. Krishnamoorthy. Interactive graph drawing on the world wide web. In *Sixth World Wide Web Conference*, 1997.

[12] P. Gajer and S. Kobourov. Grip: Graph drawing with intelligent placement. *Journal of Graph Algorithms and Applications*, 6(3):203–224, 2002.

[13] R. Hadany and D. Harel. A multi-scale method for drawing graphs nicely. *Discrete Applied Mathematics*, 113:3–21, 2001.

[14] W. He and K. Marriott. Constrained graph layout. In S. North, editor, *The 4th Internation Symposium on Graph Drawing*. LNCS 1190, 1997.

[15] G. Di Battista, P. Eades, R. Tamassia, and I. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs.* An Alan R. Apt Book. Prentice Hall, Upper Saddle River, NJ, 1999.

[16] J. Nesetril. Art of graph drawing and art. *Journal of Graph Algorithms and Applications*, 6(1):131–147, 2002.

[17] R. Paffenroth, D. Vrajitoru, T. Stone, and J. Maddocks. DataViewer: A scene graph based visualization library. In *The 5th IASTED Conference on Computer Graphics and Imaging (CGIM 2002)*, pages 200–205. ACTA Press, 2002.

[18] R. Tamassia. Constraints in graph drawing algorithms. *Constraints*, 3(1):87–120, 1998.