

# Consistent Graph Layout for Weighted Graphs

Dana Vrajitoru  
Intelligent Systems Laboratory  
Indiana University at South Bend  
Computer and Information Sciences  
danav@cs.iusb.edu

Jason DeBoni  
Intelligent Systems Laboratory  
Indiana University at South Bend  
Computer and Information Sciences  
wanderung@yahoo.com

## 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. [11], Diaz, Petit, and Serna [5]). Layouts presenting some aesthetic qualities are also appreciated (Gajer and Kobourov [9], Nesetril [12]). The problem is even more interesting and challenging when the graphs to be drawn are large (Gajer and Kobourov [9], Brandes and Wagner [3]). Another approach is to build the graph layout according to constraints that can be user-defined (Tamassia [14], He and Marriott [10]).

The best-known heuristic for generating graph layouts is certainly the spring algorithm (Eades [6]) 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.

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. [11], Eades and Kelly [8], Bodlander et al. [1]).

Among the applications of these algorithms we can cite designing electronic circuits (Battista et al. [11]), designing web sites and visualizing the content of the World Wide Web (Brandes et al. [2]), parallel computing and VLSI.

The methods we present in this paper can be seen as variations of the spring algorithm (Eades [6]) 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 the algorithms that minimize the error in the graph or attempt to find an equilibrium based on the tension forces. Section 3 presents some experimental results that validate our approach. We finish with some conclusions.

## 2. Problem Description and Algorithms

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 Euclidean 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  $P : \mathcal{V} \rightarrow \mathbf{R}^p$  that maps each vertex  $v \in \mathcal{V}$  to a geometrical point in  $\mathbf{R}^p$ , where usually  $p = 2$  or  $3$ . The edges are represented as line segments connecting the points associated with the vertices.

We will denote the undirected edges by  $uv$ , where  $u, v \in \mathcal{V}$ .

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

If  $\mathcal{V} = \{v_1, v_2, \dots, v_n\}$ , then we must find a set of points  $\{P_1, P_2, \dots, P_n\}$  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.

$$d(P_i, P_j) = W_{v_i v_j} \quad (1)$$

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:

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

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.

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

$$\forall u, v \in \mathcal{V}, \quad W_{uv} \geq 0, \quad W_{uv} = W_{vu} \quad (3)$$

$$\forall u, v, z \in \mathcal{V}, \quad W_{uz} \leq W_{uv} + W_{vz} \quad (4)$$

Equation 4 is also known as the triangulation condition. To verify this condition 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 5. This is also a necessary but not sufficient condition for the existence of the solution.

$$\forall n \in \mathbf{N}, n \geq 3, \quad \forall v_1, v_2, \dots, v_n \in \mathcal{V}, \\ w_{1n} \leq w_{12} + w_{23} + \dots + w_{(n-1)n} \quad (5)$$

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.

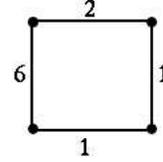


Figure 1. A graph with no solution

This problem has been proved to be NP-hard (Eades and Mendonça [7]). We would also like to consider the case where an exact solution doesn't exist and the algorithms we present will be classified according to the criteria used to find an approximate solution.

### 2.2. Minimal Total Error

To generate graphs for which a consistent layout can be found, we can start with any unweighted graph, build a layout by any method, then assign weights to the edges according to the Euclidean distance between the corresponding points.

We can also easily 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 generate the weights in a cycle of the graph such that the constraint 5 is not satisfied. This operation is linear in the selected cycle.

Let  $u$  and  $v$  be two adjacent vertices in the graph. Suppose that we have associated the geometrical points  $P_u$  and  $P_v$  with the vertices  $u$  and  $v$  respectively. Let us denote by  $err_{uv}$  the error on the edge  $uv$  computed as the difference between the weight of the edge and the Euclidean distance between the two points:

$$err_{uv} = W_{uv} - d(P_u, P_v). \quad (6)$$

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.

We would like to find a layout that minimizes the total absolute error in the graph:

$$total\_error = \sum_{\forall uv \in \mathcal{E}} |err_{uv}| \quad (7)$$

An exact solution to the graph layout presents a total error equal to 0. If the graph doesn't have an exact solution, then minimizing the total error represents a good approximation.

We will introduce next two algorithms designed to minimize the total error. 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 length of the line segment connecting them and the weight of the edge. 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 start with a totally random layout that is adjusted in a number of iteration to obtain one that is consistent. At each iteration the algorithms move 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 than 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  $u$  and  $v$  the two vertices that have been randomly selected and  $P_u$  and  $P_v$  the points assigned to them in the current layout. If the error on the edge  $uv$  is not equal to 0, we will adjust the position of the vertex  $v$  by assigning it a new point  $P'_v$  determined in the following way:

$$P'_v = P_v + \varepsilon \cdot \frac{err_{uv}}{d(P_u, P_v)} \cdot (P_v - P_u), \quad (8)$$

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

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

It can be mathematically justified that the procedure we have described reduces the error on the edge and the experiments show that the total error in the graph also decreases in

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

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 8), 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 3). 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.

If the graph has a solution, then we expect the breadth-first scan algorithm to converge faster than the random edge one. If there is no solution to the problem, we think that in some cases the random edge algorithm could find better solutions for this category of graphs.

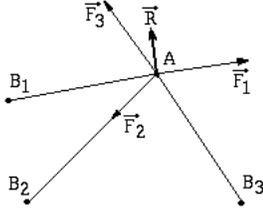
### 2.3. Equilibrium Layout

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



**Figure 2. Resulting tension force**

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.

For example, let us suppose that a vertex  $A$  is connected to three vertices  $B_1$ ,  $B_2$ ,  $B_3$  as in Figure 2.

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 6 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

$$\vec{F}_{AB} = err_{AB} \frac{\vec{AB}}{\|\vec{AB}\|} \quad (9)$$

Thus, if 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 9, 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:  $\vec{F}_{AB} = -\vec{F}_{BA}$ .

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

$$\vec{R}_A = \sum_{\forall AB \in \mathcal{E}} \vec{F}_{BA} \quad (10)$$

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:

$$P'_A = P_A + \varepsilon \vec{R}_A \quad (11)$$

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

At last, the algorithm starts again with a random layout and moves the points according to Equation 11 in a given number of iterations or until the layout converges 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.

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.

### 3. 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: (1) the unweighted graph has been generated by random; (2) we have generated a random bounded 3D layout for the graph; (3) 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 5. 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

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

Graph	Total Error			
	Initial	BFS	RE	TV
dg30	9.32e3	182.60	290.91	212.39
dg40	1.64e4	60.01	230.97	68.65
dg50	2.68e4	278.41	389.69	256.74
dg60	4.99e4	1249.94	895.73	986.75
dg70	9.95e4	780.53	998.76	208.13
dg80	1.17e5	4.39	3.50	0.13
dg90	1.34e5	6.33	1807.24	677.91
dg100	1.97e5	3437.07	4466.20	442.30
dg125	2.89e5	1732.63	2935.63	95.74
dg150	3.90e5	34.62	4.84	0.41
dg175	4.76e5	47.94	8077.07	0.47
dg200	6.70e5	7.22	0.22	0.61

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  $\epsilon$  parameter is equal to 0.005 for these results. Higher values have caused the tension vector algorithm to diverge.

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 than 3% in all of the cases, and at least half of the time less than 1%. This means that all of the algorithms have found a solution that is more than 97% precise.

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.

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 un-

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

Graph	BFS	RE	TV	TV Norm
dg30	1.48%	2.36%	1.72%	0.67
dg40	0.36%	1.39%	0.41%	0.23
dg50	0.80%	1.12%	0.74%	0.00
dg60	2.06%	1.48%	1.63%	0.00
dg70	0.64%	0.82%	0.17%	3.26
dg80	0.00%	0.00%	0.00%	0.00
dg90	0.00%	1.02%	0.38%	0.05
dg100	1.40%	1.82%	0.18%	0.45
dg125	0.46%	0.77%	0.03%	1.70
dg150	0.01%	0.00%	0.00%	0.01
dg175	0.01%	1.26%	0.00%	0.01
dg200	0.00%	0.00%	0.00%	0.01

stable 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.

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

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. Figure 3 shows these charts for the set of problems with existing solution, for the breadth-first scan, random edge, and tension vector algorithms respectively.

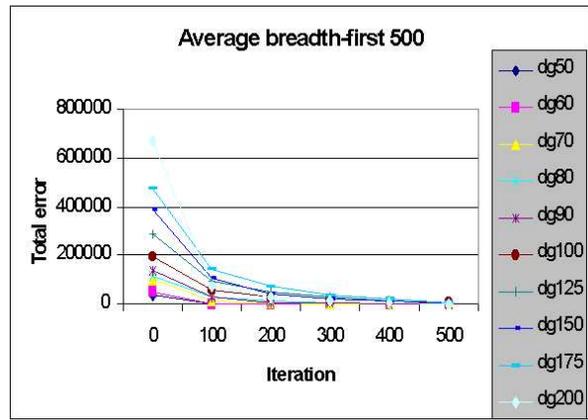
From this figure we can notice that the total error decreases very fast in the first few iterations, and then it's evolution is much slower for both sets of problems. To illustrate this phenomenon, Figure 4 shows the average total error for the second set of problems for the tension vector method in 200 iterations. This figure shows that in the 20 first itera-

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

Graph	Total Error			
	Initial	BFS	RE	TV
ukn50	8177.43	2171.06	2278.01	2082.65
ukn60	13326.01	3988.10	4115.64	3871.15
ukn70	27483.10	9714.54	9640.60	9477.52
ukn80	32838.28	11498.75	11547.96	11282.03
ukn90	38774.58	13357.58	13412.91	13106.14
ukn100	53859.34	20306.40	20338.95	19842.18
ukn125	82358.92	31329.29	31348.58	30742.93
ukn150	110082.90	42273.58	42115.45	41624.43
ukn175	136130.17	54228.63	54227.70	53380.87
ukn200	191138.27	77473.52	77285.36	76503.80

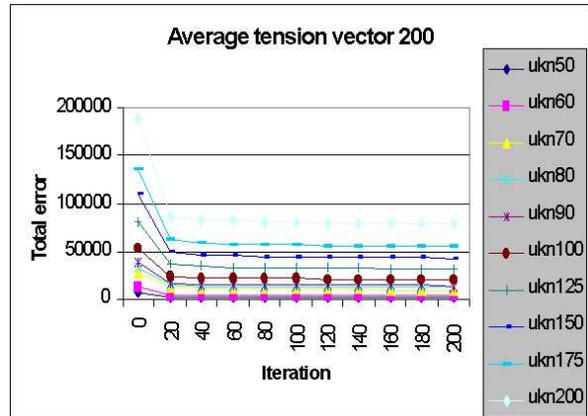
**Table 4. Total error in 1000 iterations as percentage of the total weight in the graph, non-existing solution**

Graph	BFS	RE	TV	TV Norm
ukn50	27.27%	28.62%	26.16%	0.11
ukn60	29.98%	30.94%	29.10%	0.31
ukn70	37.45%	37.17%	36.54%	8.15
ukn80	36.75%	36.91%	36.06%	8.55
ukn90	37.19%	37.35%	36.49%	7.47
ukn100	38.79%	38.85%	37.90%	8.22
ukn125	40.29%	40.31%	39.53%	8.77
ukn150	41.16%	41.01%	40.53%	8.66
ukn175	41.52%	41.52%	40.87%	9.51
ukn200	42.19%	42.08%	41.66%	12.19



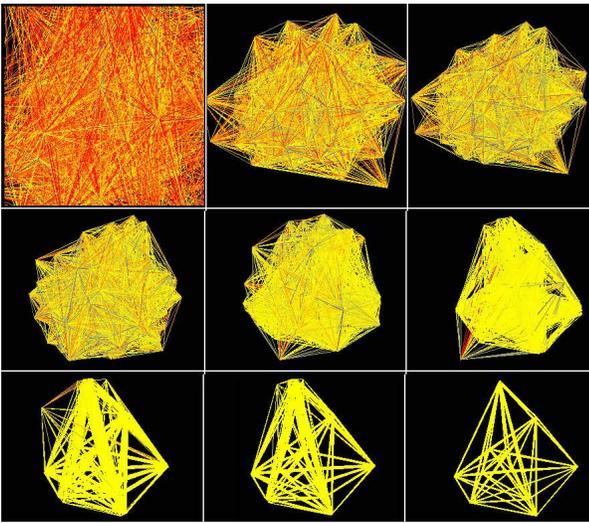
**Figure 3. Average total error for BFS in 500 iterations**

tions, the total error is adjusted a lot more than in the 180 next iterations for all of the problems.



**Figure 4. Average total error for TV in 200 iterations**

Finally, Figure 5 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 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 [13].



**Figure 5. Evolution of a graph layout in 1000 iterations with TV**

### 3.1. 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  $\varepsilon$  that are not small enough. In our example, the maximal value of  $\varepsilon$  that we could use was 0.005. This means that although the algorithm can build quite precise layouts, the limitation on the value of  $\varepsilon$  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  $\varepsilon$ . 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-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

**Table 5. Average results with the combined method in 1000 iterations**

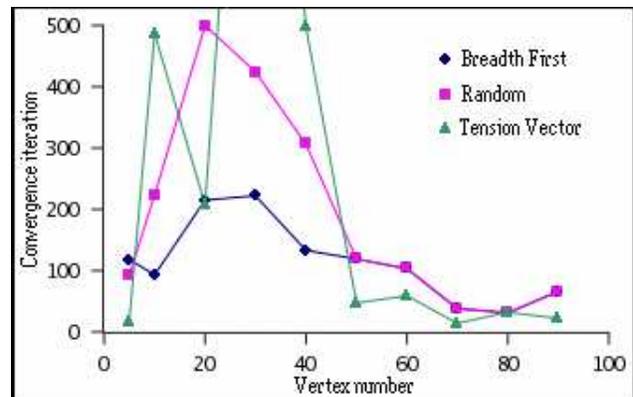
Graph	BFS	TV	Combined
dg100	634.69	442.30	213.39
dg200	0.46	0.6	0.38
ukn100	19890.52	19842.18	19614.77
ukn200	76758.73	76503.80	75779.31

combination of the two algorithms can work faster.

### 3.2. Complexity and Execution Time

The complexity of our algorithms is not easy to determine since the number of iterations is not predefined, but is chosen by hand either based on a convergence criteria or as a fixed number. All of our methods are linear over the number of edges in the graph for a single iteration. This means that in general, they are quadratic over the number of vertices in the graph.

We have performed a different experiment in which we stop the iterations when the total error has achieved a predefined lower limit, which is 0.5% in our case, meaning that the solution is at least 95% correct for an epsilon of  $\varepsilon = 0.05$ . We have used the graphs with existing solution for this experiment. Figure 6 shows the results of this experiment together with the execution time in each case, as well as the execution time for 500 iterations.



**Figure 6. Convergence iteration with a precision of 95%**

From this figure we can see that the convergence generation doesn't follow a regular pattern depending on the number of vertices. In particular, larger graph can be easier for the algorithms if they present a higher edge density

which can determine the layout more precisely. And finally, the breadth-first algorithm seems to be the one converging towards an approximately good solution faster, while the higher spikes for the tension vector method are due to the divergence occurrences we have observed.

## 4. 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 Euclidean 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. Kääh, 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] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.
- [7] 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.
- [8] P. Eades and D. Kelly. Heuristics for reducing crossings in 2-layered networks. *Ars Combin.*, 21.A:89–98, 1986.
- [9] P. Gajer and S. Kobourov. Grip: Graph drawing with intelligent placement. *Journal of Graph Algorithms and Applications*, 6(3):203–224, 2002.
- [10] W. He and K. Marriott. Constrained graph layout. In S. North, editor, *The 4th International Symposium on Graph Drawing*. LNCS 1190, 1997.
- [11] 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.
- [12] J. Neseřil. Art of graph drawing and art. *Journal of Graph Algorithms and Applications*, 6(1):131–147, 2002.
- [13] 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.
- [14] R. Tamassia. Constraints in graph drawing algorithms. *Constraints*, 3(1):87–120, 1998.