Consistent Weighted Graph Layouts

Dana Vrajitoru and Jason DeBoni

Abstract. A graph layout is a geometrical representation of a graph such that the vertexes are assigned points and the edges become line segments. In this paper we present two probabilistic algorithms that build layouts for weighted graphs such that the geometrical distances between the vertexes are consistent with the weights of the edges. Both methods start with a random layout and improve it in a number of iterations to decrease the error between the weight of the edges and the length of the corresponding line segments. Both methods have been successful in building consistent layouts with high precision.

Mathematics Subject Classification (2000). graph drawing.

Keywords. graph drawing, force-based algorithms.

1. Introduction

Suppose that several hundreds of thousands of years from now some aliens discover traces of human civilization on Earth and they try 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 find is a schedule of an airline company containing the amount of time that each flight would require to connect a given city to another. The problem is, can the aliens reconstruct the current map of the world based on that timetable?

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

Extensive work has been done on drawing unweighted graphs with the emphasis on the geometrical representation showing the structure of the graph (Battista et al. [13], Diaz, Petit, and Serna [4]) and also presenting some aesthetic qualities (Gajer and Kobourov [9], Nesetril [14]). The problem in particularly interesting

This work was supported by the IUSB Faculty Research Grant.

and challenging when the graphs to be drawn are large (Gajer and Kobourov [9], Erlingsson and Krishnamoorthy [8], Brandes and Wagner [1]). Another approach is to build the graph layout according to constraints that can be user-defined (Dornheim [3], Tamassia [16], He and Marriott [11]).

The best-known heuristic for graph layout is certainly the spring algorithm (Eades [5]) that regards the edges in the graph as springs connecting the nodes such that the springs attract the nodes if they are too far apart and repel them if they are too close. In addition, non-connected nodes repel each other. In the usual implementation, the edges are expected to have the same length. An interesting model (Branke, Bucher, and Schmeck [2]) combines this method with the use of genetic algorithms to take into account other optimization criteria like the number of edge crossing or the number of different angles in the drawing.

The methods we are presenting in this paper are inspired from the spring algorithm in which we only consider attraction forces between the vertexes. We have adapted this method for the goal of creating layouts such that there is a consistency between the distances between vertexes in the graph and the weights on the edges. We also introduce an application of genetic algorithm for the same problem.

Some research has also concentrated on weighted graphs and the best methods seems to be the force-oriented ones (Battista et al. [13], Eades and Kelly [7]). In one approach, Eades and Mendonca [6] solve the triangulation conflicts in the graph by creating copies of certain nodes to obtain not only an equilibrium layout but also one which is completely tension-free.

The methods we are presenting in this paper can largely be seen as variations of the spring algorithm (Eades [5]) in which we ignore the repulsion force exerted by non-adjacent nodes in the graph. The criteria that we are interested in is the consistency between the distances between vertexes in the graph and the weights on the edges.

The paper is structured the following way: the first section introduces the problem. The second one presents our force-based algorithms, and the third one the application of genetic algorithms to this problem. The next section presents some experimental results and we end with conclusions and with a discussion on future work.

2. The Problem

Definition. Let $G = \{\mathcal{V}, \mathcal{E}\}$ be a graph where $\mathcal{V}$ is the set of vertexes, $|\mathcal{V}| = n$, $\mathcal{E}$ is the set of edges. A *layout* for the graph is a function $P : \mathcal{V} \rightarrow \mathbf{R}^p$ 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 vertexes composing them.

Problem. Let $G = \{\mathcal{V}, \mathcal{E}, W\}$ be an unoriented, weighted graph where the weights of the edges are given by the function $W : \mathcal{E} \rightarrow \mathbf{R}^+$. We must find a layout

$P : \mathcal{V} \to \mathbf{R}^3$ such that $\forall\, u, v \in \mathcal{V},\ \mathrm{d}\,(P(u), P(v)) = W(u, v)$, where $W(u, v)$ is the weight of the edge connecting the vertexes u and v. 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 vertexes v_i and v_j, $\{v_i, v_j\} \in \mathcal{E}$, then the points associated with these vertexes are placed at a distance from each other equal to the weight of the edge.

$$\mathrm{d}\,(P_i, P_j) = W(v_i, v_j) \tag{2.1}$$

We can express the constraints in Equation 2.1 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:

$$(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2 = w_{ij} \tag{2.2}$$

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 proved to be NP-hard (Eades and Mendonca [6]).

2.1. Minimal Total Error

The minimal requirements for the graph so that there is a solution are related to the properties of the geometrical distance. Thus, if the weight of the edges represent actual distances, then they must fulfill the following conditions:

$$\forall\, A,\, B\, \in \mathcal{V}, \quad W_{AB} = W_{BA} \tag{2.3}$$

$$\forall\, A,\, B,\, C\, \in \mathcal{V}, \quad W_{AC} \leq W_{AB} + W_{BC} \tag{2.4}$$

The constraints expressed in Equations 2.3 and 2.4 represent necessary but not sufficient conditions for the existence of the solution. For example, the following graph satisfies both of these conditions, but we cannot position this graph such that the error on each edge is 0.

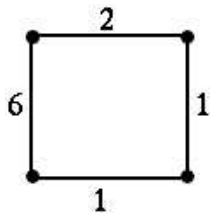

FIGURE 1. A graph with no solution

The constraints expressed in Equations 2.3 and 2.4 are a sufficient condition for the existence of the solution only in the case of a completely connected graph.

We can rewrite Equation 2.4 such that the two constraints become a sufficient condition by extending the triangular property to any closed polygon:

$$\forall n \in \mathbf{N},\ n \geq 3, \quad \forall A_1,\ A_2,\ \ldots,\ A_n \in \mathcal{V},$$
$$W_{A_1A_n} \leq W_{A_1A_2} + W_{A_2A_3} + \ldots W_{A_{n-1}A_n} \tag{2.5}$$

From Equation 2.5, we can remark that a weighted tree can always be successfully positioned.

Although an algorithm that verifies the condition 2.5 would be exponential, it is much easier to generate graphs for which we know that there is a solution. For this, we can simply assign 3D points to the vertexes in a graph and then assign to the weights to the edges the value of the distance between the two points that the edge connects.

The same way, it is easy to generate graphs for which the problem has no solution. For this, we must generate at least one cycle in the graph, and we can assign the weights in this cycle such that the constraint 2.5 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 an assignment of points to the vertices that minimizes the total absolute error in the graph. Let A and B be two vertices in the graph connected by an edge, and $P_A = (x_A, y_A, z_A)$ and $P_B = (x_B, y_B, z_B)$ the points currently assigned to them in the layout. Let us denote by err_{AB} the placement error for the edge (A, B) computed as the difference between the weight of the edge and the distance between the two points:

$$err_{AB} = W_{AB} - d(P_A, P_B). \tag{2.6}$$

Then we can express the measure of consistency for a layout as the total error in the graph computed the following way:

$$total\_error = \sum_{\forall (A,B) \in \mathcal{E}} |err_{AB}| \tag{2.7}$$

3. Force-Based Algorithms

The first category of algorithms that we're introducing start from the idea that the graph forms a dynamic system in which each element is attracted or repelled by its neighbors according to the difference between the distance between the points assigned to the nodes and the weight of the edge they compose in the graph. If the nodes are not neighbors in the graph, then they are not directly affected by each other.

3.1. Breadth-First Based Algorithm

The first algorithm consists in passing from each state of the system to another of greater probability by doing a transformation that considers only one edge of the graph at a time. At each iteration, the algorithm chooses a random vertex (origin), and then it adjusts the other points in the layout following a breadth-first traversal of the graph starting from this origin. By this method, the direct neighbors of the origin will be adjusted in the first few steps, then all of their neighbors follow, and so on. The adjustment is spreading in the graph as a wave starting from the origin.

Let A and B be two vertices in the graph such that the directed or undirected edge (A, B) is present in the graph with a weight W_{AB}. Let us suppose that the breadth-first traversal of the graph is now considering the vertex B as a neighbor of the vertex A and it must adjust its position based on the weight of the edge. Let $P_A = (x_A, y_A, z_A)$ and $P_B = (x_B, y_B, z_B)$ be the points currently assigned to the vertices A and B respectively in the layout. Equation 2.6 allows us to compute the error on this edge err_{AB}.

This error provides an estimation of how much the points are misplaced with respect to each other given 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:

$$P'_B = P_B + \varepsilon \cdot \frac{err_{AB}}{d(P_A, P_B)} \cdot (P_B - P_A), \tag{3.1}$$

where $0 < \varepsilon < 1$. In this formula, if the error is positive, then the point P_B will be moved on the line passing through P_A and P_B further away from P_A. 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 distance 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 parameter ε allows us to control the amount of adjustment that is done at each step and thus, decide on the convergence rate.

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

```
for (i=0; i<number_of_iterations; i++)
   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;
```

3.2. Tension Vector Algorithm

Let us suppose that we can construct a physical representation of the graph using some sort of interconnecting rods for the edges. Each rod 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 rods can only be deformed along the main direction. They also present an elasticity property such that when elongated, they tend to contract, and when compressed, they tend to extend. Moreover, each rod creates a contracting or extending force along the main direction proportionate to the amount of deformation that was applied to them.

We can build the graph using there rods by deforming them as necessary to fit the connections in the graph description. The physical construction would then naturally evolve to an equilibrium state in which the deformation tensions annihilate 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.

The focus now goes again to the points representing the vertices in the graph. For each point, a number of forces are applied to it as a result of the deformation along all the edges that are connected to the point. We will assume that if the composition of all the forces that apply to a vertex 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 0 local tension. We would like to find an assignment of points for the vertices of the graph such that the composition of all the deformation forces that operate on each vertex give a resulting null vector.

Additionally, we can express the optimal solution with minimal local tension based on the exact solution. We would like to find a positioning of the graph that minimizes the total norm of the resulting deformation force applied to each vertex.

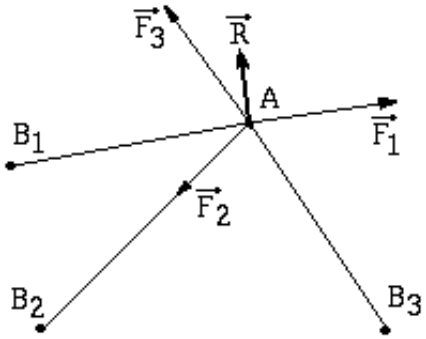

FIGURE 2. Resulting tension vector for a vertex

In the following algorithm, for each of the edges that is deformed, we will consider that an equal force is applied to each of the two vertices that are connected by the edge in opposite directions. Thus, the simple resulting force of all the deformation forces in the graph is always 0, so that the construction cannot work as a perpetuum mobile.

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 $(A, B_i) \in \mathcal{E}$, $i = 1, 2, 3$, 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 vectors representing the deformation forces $\vec{F_i}$, $i = 1, 2, 3$, we obtain the resulting force that operates on the point assigned to A, that we have 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 2.6 as the difference between the weight of the edge (A, B) 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

$$\vec{F}_{AB} = err_{AB} \frac{\vec{AB}}{\|\vec{AB}\|} \tag{3.2}$$

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 3.2, we have assumed that the deformation suffered by the edge (A, B) is equally distributed between the two points. Thus, for an unoriented graph, for each force $\vec{F}_{AB}$, there is a corresponding force equal in norm and of contrary direction applied to the other point of the edge:

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

For an oriented graph, suppose that two vertices A and B are connected by two opposite edges (A, B) and (B, A), of possible different weights. In this case, the force applied to each point will be the average of the deformation forces resulting from each edge. 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:

$$\vec{R}_A = \begin{cases} \sum_{\forall (A,B) \in \mathcal{E}} \vec{F}_{BA}, & \text{for an unweighted graph} \\ \sum_{\forall (A,B),(B,A) \in \mathcal{E}} \vec{F}_{(B)A}, & \text{for a weighted graph} \end{cases} \tag{3.3}$$

If P_A is the point associated to the vertex A at a particular moment and $\vec{R}_A$ the force operating on it, the algorithm assumes that at the next moment the point would have moved to a new location P'_A defined as follows:

$$P'_A = P_A + \varepsilon \vec{R}_A \tag{3.4}$$

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

At last, the algorithm with start again with a random assignment of points to the vertices in the graph and will move them according to the rule described in Equation 3.4 in a number of iteration until the points convergence to an equilibrium. The following is a general pseudocode description of the algorithm:

```
for (i=0; i<number_of_iterations; i++)
   for all A in V
      compute RA;
   for all A in V
      PA = PA + epsilon * RA;
```

It is important to remark that in the new algorithm, the resultant force for every point of the graph is computed based on the current assignment of points before any of them is moved. The new points are computed after all the resulting forces are determined. This is a major difference between this algorithm and the ones introduced in Section 3, in which each point was moved in one step, and the next point to be moved was based on the new position of all of the previous points.

4. Genetic Algorithms for Graph Layouts

In this section we present an application of genetic algorithms (GAs) to our graph layout problem.

A genetic algorithm deriving graph layouts represents an alternative to the algorithms presented in the previous section. The advantage of the GA is that the choice of the fitness function can allow us to express some geometrical constraints that are not easy to achieve by the force-based methods, like maximizing the enclosed volume, or even aesthetic qualities of the solutions. Although this will be the object of future research, the results obtained so far seem promising. The difficulty of this method so far is that we expect it to generate accurate solutions much slower than the other algorithms.

4.1. The Genetic Algorithms

The GAs are probabilistic algorithms generally used for optimization problems. Through operations inspired from the natural selection, they search for the best solution to a problem (Holland [12], Goldberg [10]).

Given a search space E, we must find an element $e \in E$ maximizing a performance mapping f defined on E, called *fitness*. The elements of E are called chromosomes and each of them represents a potential solution to the problem. To apply a GA, the chromosomes must be coded as a sequence of genes, the most often binary. The position of a given gene in the chromosome sequence is called locus.

The general GA starts with an initial population containing a number of chromosomes, $size$, and representing the first generation. Given an old generation, a new generation is built from it using the following operations.

The *selection* chooses $size$ chromosomes from the old generation, according a better chance to chromosomes presenting a better performance. The chance of each chromosome to be selected is proportional to its fitness. This method is generally known as the *roulette wheel* or *fitness-proportionate* selection.

The second operation is the *crossover*, generally employed to combine the genes of two chromosomes, known as parents, with the hope of constructing better ones, known as children. We have chosen the *uniform* crossover (Syswerda [15]) for our application. This operator decides independently for each locus from 1 through $L-1$, whether the parents genes will be swapped or not, with a given probability. We have adopted the swap probability of 0.5 for our research.

The third operation is the *mutation* which decides randomly for every locus, if the gene must be mutated to be replaced with a different value. If the genes are binary, then the mutation will replace a gene with its opposite. This decision also depends on a given probability which is in general relatively low.

Finally, some individuals from the old generation can be cloned with no modification into the new generation, operation known as *reproduction*. In our case, we have chosen to reproduce the best chromosome from the old generation,

if the new generation hasn't produced any chromosome of higher fitness. This operation insures the monotony of the algorithm (Vrajitoru [17]).

The GA consists in building new generations until a stop condition is fulfilled (usually the population convergence) or until a given number of generations is reached. In our case we have run all of our experiments up to 1000 generations.

4.2. Genetic Representation of a Graph Layout

The first challenge of using the GAs to produce consistent graph layouts is representing a graph layout as a chromosome. If the graph itself is known to the algorithm as a global object that we do not need to include in each chromosome, then a graph layout is simply a sequence of points, each of them described by three real coordinates. We can represent a real number as a sequence of binary genes by a discretization, but we have chosen to allow the genes themselves to be real numbers. Thus, the size of a chromosome is three times the number of vertices in the graph. Considering that this number is generally much lower than the number of edges, it allows us to have a comfortable number of chromosomes by generation without running into memory problems.

The only operation that must be adapted to this problem is the mutation. Since a real number doesn't have an obvious opposite, we simply chose to replace it by mutation with another real number chosen randomly in a given bounding box.

The next choice is the fitness function, which should represent the total error in the graph. Since the algorithms usually expect a fitness that increases as the solutions get better, our fitness is expressed by

$$f(layout) = \frac{1}{1 + TotalError(layout)}$$

The maximum for this function is 1, but we chose to show the total error in the resulting best chromosome in the last generation instead of the actual fitness.

5. Experimental Results

There are two distinct sets of problems that we will have to deal with.

In the first category, we have graphs for which there exists a consistent layout. For this category, we expect the tension vector algorithm to converge faster to the solution. We have used 11 graphs in this category, with the number of vertices going from 50 to 200.

In the second category, we have problems for which there is no consistent layout. We have chosen 11 problems in this category, with the number of vertices the same as before. 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 expect the breadth-first based algorithm to find better solutions for this category of problems.

TABLE 1. Average total error in 1000 iterations for graphs with existing solution

♯Vertices	Total Weight	BF	TV	GA
50	34751.2	4.18	8.33	32602.1
60	60600.7	27.56	25.2587	56821.0
70	121708	1.15	1.29	114702.0
80	141513	0.00169601	0.00033846	132823.0
90	176463	0.00195465	0.000355982	166298.0
100	245876	14.0295	77.2433	230943.0
125	379287	0.00372101	117.348	357410.0
150	498501	0.00531211	92.9607	469440.0
175	639429	0.00628272	129.093	602553.0
200	900704	0.00943293	171.594	849261.0

TABLE 2. Average total error in 1000 iterations for graphs with non-existing solution

♯Vertices	Total Weight	BF	TV	GA
50	7959.8	41.15	80.23	5243.71
60	13300.5	77.44	190.11	8977.52
70	25938.8	187.03	225.75	17664.50
80	31289.1	219.66	262.29	21431.80
90	35912.3	257.89	nan	25014.3
100	52355.1	398.76	619.81	36344.0
125	77765.8	616.54	836.72	54445.2
150	102694	833.31	1073.74	72310.0
175	130597	1066.42	1536.99	92783.7
200	183641	1529.38	43.0	131198.0

Tables 1 and 2 present the results of the 3 algorithms for the graphs with existing and non-existing solution respectively. In the case of the force-based methods, these represent 1000 iterations with $\varepsilon = 0.05$. In the case of the genetic algorithms, they represent 1000 generations with a population size of 50 chromosomes. All of the results are an average over 50 trials with the same parameters but different sequences of random numbers.

The first column in each table indicates the number of vertices in the graph. The second column represents the sum of the weights on all the edges in the graph, which gives us an indication of the precision of the layouts produced by each method. The *nan* entry in one of the cells indicates that the algorithm has diverged on all the trials for that particular graph. Thus, the tension vector algorithms shows some occasional divergence problems.

From these tables we can see that the force-based methods seem to be performing much better than the genetic algorithms. These methods are also faster under the described experimental conditions. The genetic algorithms would probably require a larger number of generations to produce comparable results, or more specific operators, like a mutation that uses one of the force-based methods to find a better placement for a point in the layout. Perhaps a combination of the genetic algorithms as a first step with a force-based method to refine the solution would produce even better results.

Finally, Figure 3 shows the evolution of a layout for a graph with 125 vertices through 1000 iterations of the tension vector algorithm. The edges colored in red are shorter than they should be, those colored in blue are longer than they should be, and a yellow edge has exactly the length indicated by its weight. The configuration of weights for this graph constraints the vertices to be organized in clusters in the layout, which can be observed in the final layout.

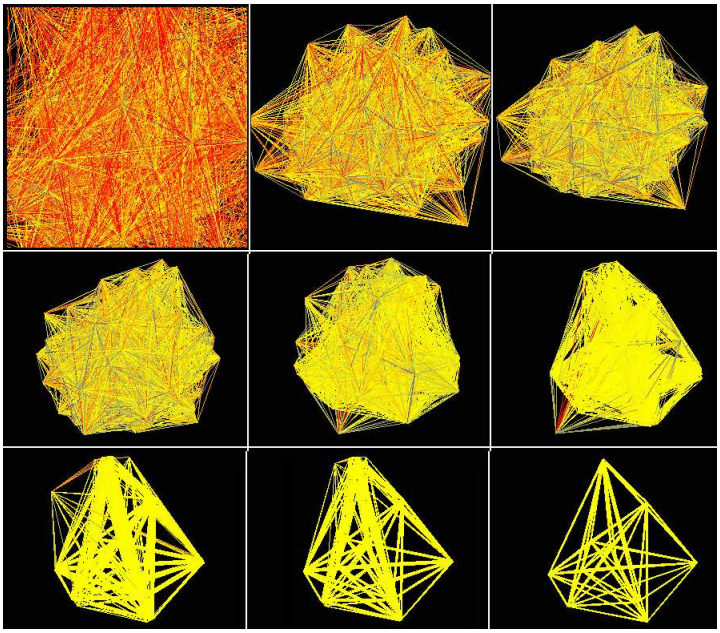

FIGURE 3. Evolution of a graph layout in 1000 iterations with TV

6. Conclusion

In this paper we have introduced three algorithms that aim to construct graph layouts that are consistent with the weights in the graph. The first two algorithms are force-based and inspired by the spring algorithm. They build incrementally

better layouts starting from a random layout by considering attraction and repulsion forces between vertices generated by the error on the edges. The third one is an application of genetic algorithms to this particular problem with a uniform crossover and a random mutation.

The first two methods have produced solutions with reasonable precision. Generally, the breadth-first method seems to be more precise. The genetic algorithms have shown a much slower evolution than the force-based methods, but they show promise because the fitness function could incorporate other geometrical constraints than simply a low total error.

References

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

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

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

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

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

[6] 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.

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

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

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

[10] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading (MA), 1989.

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

[12] J. H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, 1975.

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

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

[15] G. Syswerda. Uniform crossover in genetic algorithms. In J. D. Schaffer, editor, *Proceedings of the International Conference on Genetic Algorithms*, San Mateo (CA), 1989. Morgan Kaufmann Publishers.

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

[17] D. Vrajitoru. Genetic programming operators applied to genetic algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 686–693, Orlando (FL), 1999. Morgan Kaufmann Publishers.

Dana Vrajitoru
Intelligent Systems Laboratory
Indiana University South Bend
Computer and Information Sciences
South Bend, IN, USA
e-mail: `danav@cs.iusb.edu`

Jason DeBoni
Intelligent Systems Laboratory
Indiana University South Bend
Computer and Information Sciences
South Bend, IN, USA
e-mail: `wanderung@yahoo.com`