JIGGLE: Java Interactive Graph Layout Environment

Daniel Tunkelang

Carnegie Mellon University

quixote@cmu.edu

Abstract. JIGGLE is a Java-based platform for experimenting with numerical optimization approaches to general graph layout. It can draw graphs with undirected edges, directed edges, or a mix of both. Its features include an implementation of the Barnes-Hut tree code to quickly compute inter-node repulsion forces for large graphs and an optimization procedure based on the conjugate gradient method. JIGGLE can be accessed on the World Wide Web at http://www.cs.cmu.edu/~quixote.

1. Introduction

Most general graph drawing algorithms fall into two categories: algorithms based on physical models, and hierarchical algorithms that place nodes on discrete layers [BETT94].

The physically based algorithms focus on undirected graphs [Ea84, KK89, FR91, FLM94, SM94, Tu94, Ig95, CP96, DH96]. In most of the proposed models, straight-line edges attract their endpoints according to a spring law, and nodes repel each other as if they were like-charged particles. These algorithms use numerical optimization to obtain a drawing whose "energy" is locally minimal.

In contrast, the hierarchical algorithms focus on directed graphs—generally graphs that are acyclic or that can be made acyclic by the reversal of a few edges [STT81, GKNV93]. These algorithms assign nodes to discrete layers so that most edges are directed from lower layers to higher ones, and then permute the nodes on each layer to minimize edge crossings. When edges traverse more than one layer, the algorithms introduce "dummy nodes" on the intermediate layers that induce bends along the edges.

JIGGLE provides a framework for experimenting with numerical optimization approaches to drawing graphs with undirected edges, directed edges, or a mix of both. In addition to incorporating the key aspects of the physically based algorithms, JIGGLE implements three significant improvements. The first is to incorporate directed edges into the physical model traditionally applied to undirected graphs. The second is to use the Barnes-Hut tree-code to compute repulsion forces in $(n \log n)$ time. The third is to adapt the conjugate gradient method to perform the numerical optimization.

S.H. Whitesides (Ed.): GD'98, LNCS 1547, pp. 413-422, 1998.
Springer-Verlag Berlin Heidelberg 1998

2. Graph Drawing as Numerical Optimization

The physically based algorithms for undirected graphs treat graph drawing as a numerical optimization problem. A two-dimensional drawing of an n node graph corresponds to a point in R^{2n}, and the goal is to produce a drawing that is locally minimal with respect an objective function that quantifies the desired aesthetic criteria. Given the difficulty of general optimization, these algorithms usually restrict themselves to objective functions that are continuous and differentiable.[1] This restriction allows them to use iterative first-order optimization techniques that depend solely on the gradient of the objective function. Often these algorithms do not even specify an objective function explicitly; rather, they specify its gradient (actually, the negative gradient) in terms of "force laws."

Usually, the laws include spring forces that attract the endpoints of edges towards each other and repulsion forces that push all nodes away from each other. For example, Fruchterman and Reingold use the following force laws: $f_{spring}(d) = d^2/k$; $f_{repulsion}(d) = -k^2/d$. The constant k is a pre-specified optimal edge length; when an edge is of length k, its endpoints exert no net force on each other.

The hierarchical algorithms for directed graphs take a markedly different approach. First, they assign nodes to discrete layers and create dummy nodes so that all edges connect nodes or dummy nodes on consecutive layers. They then permute the nodes on each layer to avoid edge crossings. Minimizing the number of edge crossings, however, is an NP-hard discrete optimization problem [GJ83]. Hence, the algorithms avoid edge crossings by sorting nodes and dummy nodes according to the mean or median positions of their neighbors on adjacent layers. The algorithms perform the sorting iteratively. First, they traverse the layers from first to last, sorting the nodes on each layer according to the positions of the nodes' neighbors on the previous layer. Then, they reverse this process, sorting nodes according to the positions of the nodes' neighbors on the following layer. They alternate between forward and reverse traversals until meeting some convergence criterion. After they have established the order of nodes on each layer, they optimize node positions subject to that order.

We could view the hierarchical drawing approach in terms of constrained numerical optimization. Our objective function might be the sum of the squares of horizontal components of edges, so that a particular node's contribution to the objective function would be minimal when its x coordinate is the average of the x coordinates of its neighbors. We would need two sets of constraints: the y coordinates would be fixed according to the layer assignments, and the nodes on each layer would have to be separated from each other by some minimum distance.

---

[1] Most of the objective functions have singularities when two nodes occupy the same coordinates, but they handle these singularities by randomly separating the coincident nodes.

Unfortunately, every possible ordering of nodes on the layers would lead to a drawing that is a constrained local minimum. The constraints separate all of the local minima from each other, necessitating an optimization procedure that can jump over the constraint barriers.

Instead of attempting such an approach, we use an unconstrained approach, replacing the minimum separation constraints with repulsion forces.

3. Our Physically Based Model for Mixed Graphs

Since the spring and charge model works well for undirected graphs, we looked for a simple modification that would take into account directed edges. The spring force for an undirected edge tries to minimize the distance between the endpoints. Inspired by the use of layers in the hierarchical approach, we made the following change in the spring law: when a spring is associated with a directed edge, it tries to place the "to" endpoint a distance proportional to k below the "from" endpoint.

This way of including directed edges in the model works surprisingly well. It does not, however, behave reasonably if the graph has a directed cycle. To handle this situation, we borrow from the hierarchical algorithms: we preprocess the graph using depth-first traversal to detect directed cycles and break the cycles by reversing the back edges of the traversal. Having solved this problem, we discovered that it was necessary to introduce a short-range repulsion force between nodes and edges to prevent nodes from being placed on top of edges. We use an inverse-square law for node-edge repulsion.

The JIGGLE interface allows the user to choose among a variety of force laws. The spring force can be logarithmic, linear. or quadratic. The user can set the short-range ($d \quad 2k$) and long-range ($d > 2k$) repulsion forces independently as inverse-linear or inverse square laws. The user can also turn off long-range repulsion forces, as Fruchterman and Reingold do in their "grid variant" approach. The node-edge repulsion force is also an option that the user can toggle.

4. Computing the Forces Efficiently

Our straightforward computation of the spring forces requires (m) time. Beating this time would require some way of filtering or summarizing edges—a strategy we do not consider here.

We focus instead on computing the repulsion forces efficiently. Since there are $\frac{1}{2}n(n-1)$ node pairs, a straightforward computation of node-node repulsion forces would require (n^2) time. If the graph is sufficiently dense (i.e. m is (n^2)), there is little point in speeding up this computation, since the spring computation will take

(n^2) time. Often, however, we are interested in large, sparse graphs. For these graphs, we can benefit significantly by reducing the computational cost of repulsion forces from quadratic to linear or near-linear time.

We have implemented Fruchterman and Reingold's "grid variant" approach. This heuristic ignores repulsion forces between nodes that are more than $2k$ apart in the drawing. If the drawing distributes the nodes sufficiently uniformly, then the computation of repulsion forces requires only (n) time. Unfortunately, the abrupt cut-off of repulsion forces can significantly distort the drawing and cause the optimization procedure to oscillate around the cut-off boundaries. Also, if the node distribution is highly non-uniform, then the computational cost may be as high as (n^2).

In order to reduce the computational cost for repulsion forces without significant loss of accuracy, we have implemented the Barnes-Hut tree-code [BH86]. Every time we compute the repulsion forces, we insert the nodes into a quad-tree. The root cell of the quad-tree represents the bounding rectangle of the drawing space. When we insert the first node, we split this root into four children cells which represent the four quadrants of the rectangle. Subsequent insertions lead us to split these cells and their descendants so that each leaf cell contains at most one node. We then traverse the nodes to compute the repulsion force acting on each node. First, we look at the node's siblings in the tree, then at its parent's siblings, then the grandparent's siblings, and so forth. We compute the repulsion force between a node and a cell as follows: if the cell is itself a leaf, then we use the regular repulsion law for two nodes. If the cell is not a leaf, then we determine if the node's rectangle and the cell's rectangle intersect. If the rectangles share even a single point, we recursively compute the repulsion between the node and the cell's children. Otherwise, we compute the repulsion force as if all of the cell's nodes were concentrated at their centroid.

The running time for Barnes-Hut depends on the height of the tree and the number of cells than a leaf cell can intersect. Assuming that nodes are never assigned to identical coordinates, we can bound the height of the tree by the number of bits used to represent a point in the drawing space, which we assume to be $O(\log n)$. In order to bound the number of cells that a leaf cell can intersect by a constant, we can rebalance the quad-tree by splitting cells after each insertion [Mo95]. If we perform this rebalancing, then the overall running time for Barnes-Hut is $(n \log n)$. In practice, the overhead of rebalancing the tree does not seem to justify the theoretical performance guarantee. We therefore use Barnes-Hut unbalanced quad-trees, relying on its observed $(n \log n)$ behavior.

Computing node-edge repulsion naively would take (nm) time—an even more expensive operation than that of computing node-node repulsion! Here, however, we can take advantage of node-edge repulsion's being a short-term force, and use a grid-variant technique to address it. If the average edge length is (k) and the node distribution is sufficiently uniform, then we can compute node-edge repulsion forces

in (m) time. In order to avoid the cut-off problem, we subtract a constant from the magnitude of the force, thereby making it continuous. We could instead use a variation of Barnes-Hut, but we have not implemented such an approach.

In summary, we compute the gradient in $(m + n \log n)$ time. For very large sparse graphs, we might consider using the Fast Multipole Method, which computes repulsion forces in (n) time [GR87]. Its overhead, however, is too large to make it practical for graphs of less than 10^4 nodes.

5. The Optimization Procedure

Quantifying the aesthetics as force laws and computing the forces efficiently is only half of the problem. We also need a procedure that computes a drawing that is locally optimal with respect to those force laws.

As we noted in the introduction, the force laws actually determine the negative gradient of an implicit objective function. Knowing the gradient allows us to use iterative first-order optimization strategies. Most of the published algorithms use the method of steepest descent or some variation thereof. Steepest descent always chooses the negative gradient as a search direction, and then uses some sort of line search procedure to determine how far to move along that direction. If the objective function has a lower bound and the line search satisfies a few reasonable assumptions, then the method of steepest descent will always converge to a local minimum. Its convergence rate, however, is linear, and can be very poor in practice. In order to obtain better performance, we have adapted the conjugate gradient method, which has superlinear convergence in theory and outperforms steepest descent in practice. We refer the reader to a discussion and analysis of these and other first-order optimization techniques in any standard optimization text, such as Gill, Murray, and Wright [GMW81].

The conjugate gradient method is an iterative method that finds the unique global minimum of a quadratic function when its Hessian matrix (which is constant) is positive definite. Each iteration of the conjugate gradient method performs a line search; if this line search is exact, then the method will solve the minimization problem exactly in n iterations, where n is the number of variables in the minimization problem. The first iteration searches along the negative gradient; subsequent iterations use a linear combination of the previous search direction and the current gradient.

Our problem, unfortunately, is somewhat messy. The objective function is certainly not quadratic, nor is its Hessian matrix positive definite. Not only does it not have a unique global minimum, but it will often have many local minima. Moreover, finite-precision arithmetic rules out an exact line search in principle, and computational expense limits the accuracy of the search in practice. Nonetheless, we

can use the conjugate gradient method on a non-quadratic objective function with an inexact line search, as long as we restart it from scratch whenever the current search direction is no longer a descent direction.

If a function is quadratic and its Hessian is positive definite, then we can make a theoretical comparison of the convergence behaviors of the steepest descent and conjugate gradient methods. In both cases, the rate of convergence depends on the spectral condition number   of the Hessian—that is, the ratio between the largest and smallest eigenvalues. If we use steepest descent, then, as we approach the solution, each iteration only reduces the distance to it by a factor of $((\ -1)\ /\ (\ +1))^2$. For the conjugate gradient method, this factor becomes $((\quad -1)\ /\ (\quad +1))^2$. An elegant analysis of the convergence behavior of each method appears in Shewchuk's introduction to the conjugate gradient method [Sh94].

Since the function is not quadratic and our line search is inexact, we cannot make any guarantees about the rate of convergence. Nonetheless, our empirical results lead us to conjecture that our variation of the conjugate gradient method still provides an asymptotic improvement on the number of iterations necessary for convergence.

Having a method to compute the search direction, we still need to determine the size of the step we take in that direction. The cheapest method—a constant step size—has the problem that too large a step size can lead to non-convergent oscillation, while too small a step size results in a very large number of iterations before convergence. The other extreme is to use a very accurate line search, e.g. one based on polynomial interpolation. Such an approach, however, has the drawback that each line search may require many gradient evaluations. Our approach uses an adaptive step size. We us an empirically determined initial step size, and we increase or decrease this step size on each iteration based on the previous one. We perform a gradient evaluation to ensure that we do not overshoot; if our step size is acceptable, we use this computed gradient for the following iteration.

6. Empirical Results

In order to measure the effects of introducing Barnes-Hut and the conjugate gradient method, we have compiled empirical results for some undirected graphs. Our test suite of square meshes, complete binary trees, and hypercubes consists of graphs that we could easily parameterize by size and for which we could confirm the quality of the drawings by eye. While we cannot claim that our observations generalize to all graphs or even to all undirected graphs, we are at least able to venture a few conjectures.

The table below shows the empirical results we obtained using a PC with a 133 MHz Pentium processor running Microsoft's JVM under Windows 95. We drew each graph under four conditions: with and without Barnes-Hut, and with either

conjugate gradients or steepest descent. We held all other parameters constant: k = 100, quadratic spring force, inverse-linear short and long range node-node repulsion, and no node-edge repulsion. We produce the initial drawing by scattering the nodes randomly on an 800 by 600 rectangle. For each graph and setting, we ran seven tests and used the median for each measured quantity: the number of iterations, the number of gradient evaluations, and the total time necessary to converge to an optimal drawing.

We used a conservative convergence criterion: the average of the square of the distances moved by the nodes on an iteration must be less than 0.01. Often we could have stopped the algorithm much sooner, but we did not want to introduce a subjective element by using an "eyeball" convergence norm.

Several entries in the table are marked with asterisks (*); for these, the time necessary to converge was over half an hour.

			Conj. Gradient. with Barnes-Hut			Conj. Gradient w/o Barnes-Hut			Steepest Descent with Barnes-Hut			Steepest Descent w/o Barnes-Hut		
Graph	n	m	# of iters	grad evals	time (secs)	# of iters	grad evals	time (secs)	# of iters	grad evals	time (secs)	# of iters	grad evals	time (secs)
mesh	64	112	82	121	0.91	73	111	0.50	165	241	1.74	197	273	1.36
mesh	144	264	129	208	4.50	111	183	3.39	365	534	10.89	390	570	13.12
mesh	256	480	190	327	14.65	173	266	15.67	683	1028	46.04	554	800	58.17
mesh	400	760	236	405	30.33	211	352	52.32	1148	1725	122.47	694	1020	173.74
mesh	576	1104	487	864	104.04	335	591	170.04	1403	2108	236.45	1223	1835	647.83
mesh	784	1512	598	1059	177.93	453	825	468.08	2603	3908	639.49	*	*	*
mesh	1024	1984	535	906	233.43	506	918	829.43	2147	3224	775.18	*	*	*
tree	63	62	145	272	1.67	166	274	1.04	485	706	5.32	702	996	4.71
tree	127	126	414	714	11.81	481	813	11.89	1985	2970	44.80	1627	2418	44.73
tree	255	254	1332	2454	104.91	1146	2102	126.14	4741	7115	266.96	4764	7144	529.50
tree	511	510	2575	4777	557.37	1961	3852	900.99	*	*	*	*	*	*
cube	64	192	86	142	1.26	64	98	0.45	73	115	0.99	75	110	0.66
cube	128	448	97	171	3.46	101	168	2.75	110	171	3.43	100	151	3.30
cube	256	1024	115	216	9.84	99	160	9.56	133	206	9.76	101	159	12.80
cube	512	2304	128	238	28.64	121	205	50.32	116	180	21.09	15	233	67.80
cube	1024	5120	141	277	77.48	128	216	226.48	163	252	71.29	154	235	289.53

The data in the table allow us to make several observations:

1) Denser graphs require fewer iterations than sparser ones. Intuitively, we can see that for denser graphs, where the gradient is dominated by the spring terms, the objective function behaves much more linearly than it does for sparser ones, where the repulsion terms play a more significant role.

2) The average number of gradient evaluations per iteration is less than 2, suggesting that the adaptive step-sizing works well for both the conjugate gradient and steepest descent procedures.

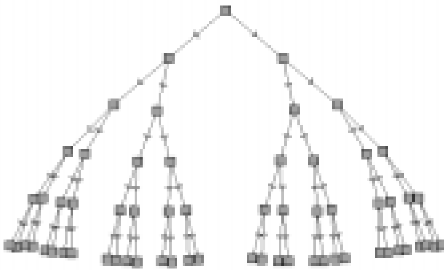
3) Barnes-Hut is a major improvement over the naïve computation of repulsion forces for large graphs. It is slightly more expensive for small ones; the break-even point seems to be between 100 and 200 nodes, depending on the topology of the graph. The inaccuracies of Barnes-Hut seem to require slightly more iterations for convergence, but the decreased cost of an iteration more than makes up for the increased number of them.

4) For the meshes and trees, the conjugate gradient method requires far fewer iterations than steepest descent. We conjecture that, for most graphs, the number of iterations for the conjugate gradient method is sublinear in the number of nodes, while the number for steepest descent is linear.

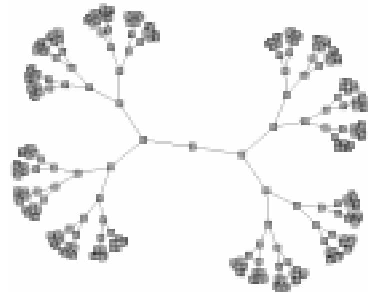
5) For the hypercubes, there seems to be no clear winner among the two methods. In fact, the number of iterations does not seems to grow with the number of nodes. It is not clear whether this rapid convergence is the result of their logarithmic density or their high degree of symmetry.

7. Examples

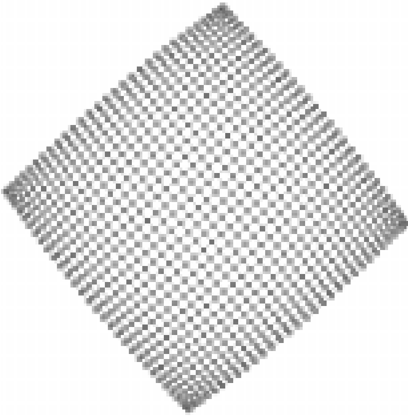
The following are examples of drawings produced by JIGGLE using the default settings. For each drawing, we indicate the number of iterations and the amount of time used to produced it.



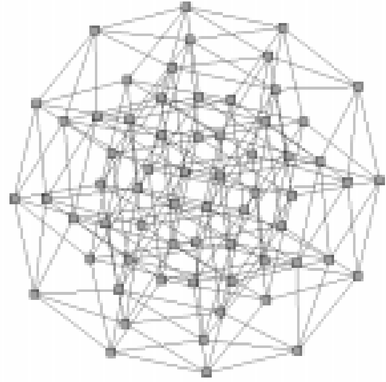
63-Node Directed Binary Tree
300 iterations, 5 secs



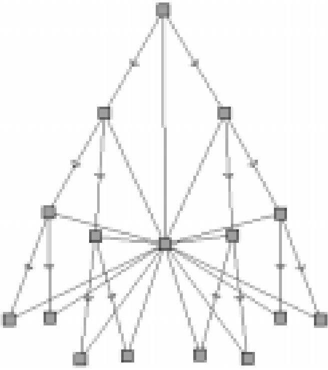
255-Node Undirected Binary Tree
900 iterations, 65 secs



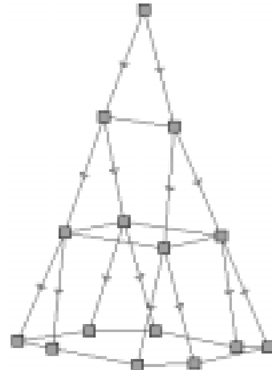
32 x 32 Square Mesh
300 iterations, 135 secs



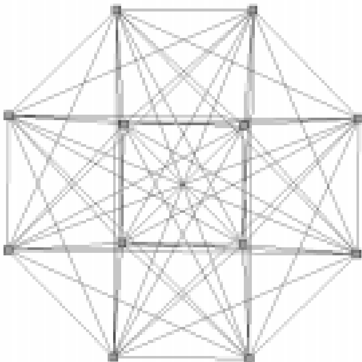
6-Dimensional Hypercube
80 iterations, < 1 sec



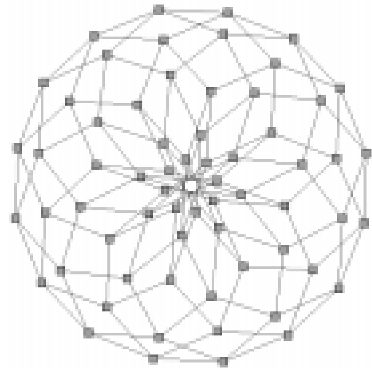
16-Node Binary Tree/Star
100 iterations, < 1 sec



16-Node Binary Tree with Level Cycles
100 iterations, < 1 sec



Complete Graph of 12 Nodes
50 iterations, < 1 sec



8 by 8 Torus
100 iterations, < 1 sec

8. Future Work

There are several directions in which we hope to extend the present work. The first is to generalize the force laws, taking into account nodes with width and height. The second is to make the objective function time-dependent. As we have seen, some objective functions are ideal for the early iterations, while other are more suitable for fine-tuning. We hope to develop a formal approach to "scheduling" the gradient as a function of the number of iterations. The third is to improve over-all efficiency. We believe that our optimization procedures can be better tuned, and we are also developing an implementation of the JIGGLE algorithms in C++.

References

[BETT94] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, "Algorithms for Drawing Graphs: An Annotated Bibliography," *Computational Geometry: Theory and Applications*, vol. 4, pp. 235-282, 1994.

[BH86] J. Barnes and P. Hut, "A Hierarchical O(N log N) force-calculation algorithm," *Nature*, vol. 324, pp. 446-449, 1986.

[CP96] M. Coleman and D. Parker, "Aesthetics-based Graph Layout for Human Consumption," *Software—Practice and Experience*, vol. 26, pp. 1415-1438, 1996.

[DH96] R. Davidson and D. Harel, "Drawing Graphs Nicely Using Simulated Annealing," *ACM Transactions on Graphics*, vol. 15, no. 4, pp. 301-331, 1996.

[Ea84] P. Eades, "A Heuristic for Graph Drawing," *Congressus Numerantium*, vol. 42, pp. 149-160, 1984.

[FLM94] A. Frick, A. Ludwig, and H. Mehldau, "A Fast Adaptive Layout Algorithm for Undirected Graphs," in *Proceedings of Graph Drawing '94*, pp. 388-403, 1994.

[FR91] T. Fruchterman and E. Reingold, "Graph Drawing by Force-Directed Placement," *Software—Practice and Experience*, vol. 21, no. 11, pp. 1129-1164, 1991.

[GJ83] M. Garey and D. Johnson, "Crossing Number is NP-Complete," *SIAM Journal on Algebraic and Discrete Methods*, vol. 4, no. 3, pp. 312-316, 1983.

[GKNV93] E. Gansner, E. Koutsofios, S. North, and K. Vo, "A Technique for Drawing Directed Graphs," *IEEE Transactions on Software Engineering*, vol. 19, no. 3, 1993.

[GMW81] P. Gill, W. Murray, and M. Wright, *Practical Optimization*, Academic Press, London, 1981.

[GR87] L. Greengard and V. Rokhlin, "A Fast Algorithm for Particle Simulations," *Journal of Computational Physics*, vol. 73, pp. 325-348, 1987.

[Ig95] J. Ignatowicz, "Drawing Force-Directed Graphs using Optigraph," in *Proceedings of Graph Drawing '95*, pp. 333-336.

[KK89] T. Kamada and S. Kawai, "An Algorithm for Drawing General Undirected Graphs," *Information Processing Letters*, vol. 31, pp. 7-15, 1989.

[Mo95] D. Moore, "The Cost of Balancing Generalized Quadtrees," Technical Report COMP TR95-246, Rice University, Department of Computer Science, 1995.

[Sh94] J. Shewchuk, "An Introduction to the Conjugate Gradient Method without the Agonizing Pain," Carnegie Mellon University, School of Computer Science, unpublished draft.

[SM94] K. Sugiyama and K. Misue, "A Simple and Unified Method for Drawing Graphs: Magnetic-Spring Algorithm," in *Proceedings of Graph Drawing '94*, pp. 364-375.

[STT81] K. Sugiyama, S. Tagawa, and M. Toda, "Methods for Visual Understanding of Hierarchical System Structures," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 11, no. 2, 1981.

[Tu94] D. Tunkelang, "A Practical Approach to Drawing Undirected Graphs," Technical Report CMU-CS-94-161, Carnegie Mellon University, School of Computer Science, 1994.